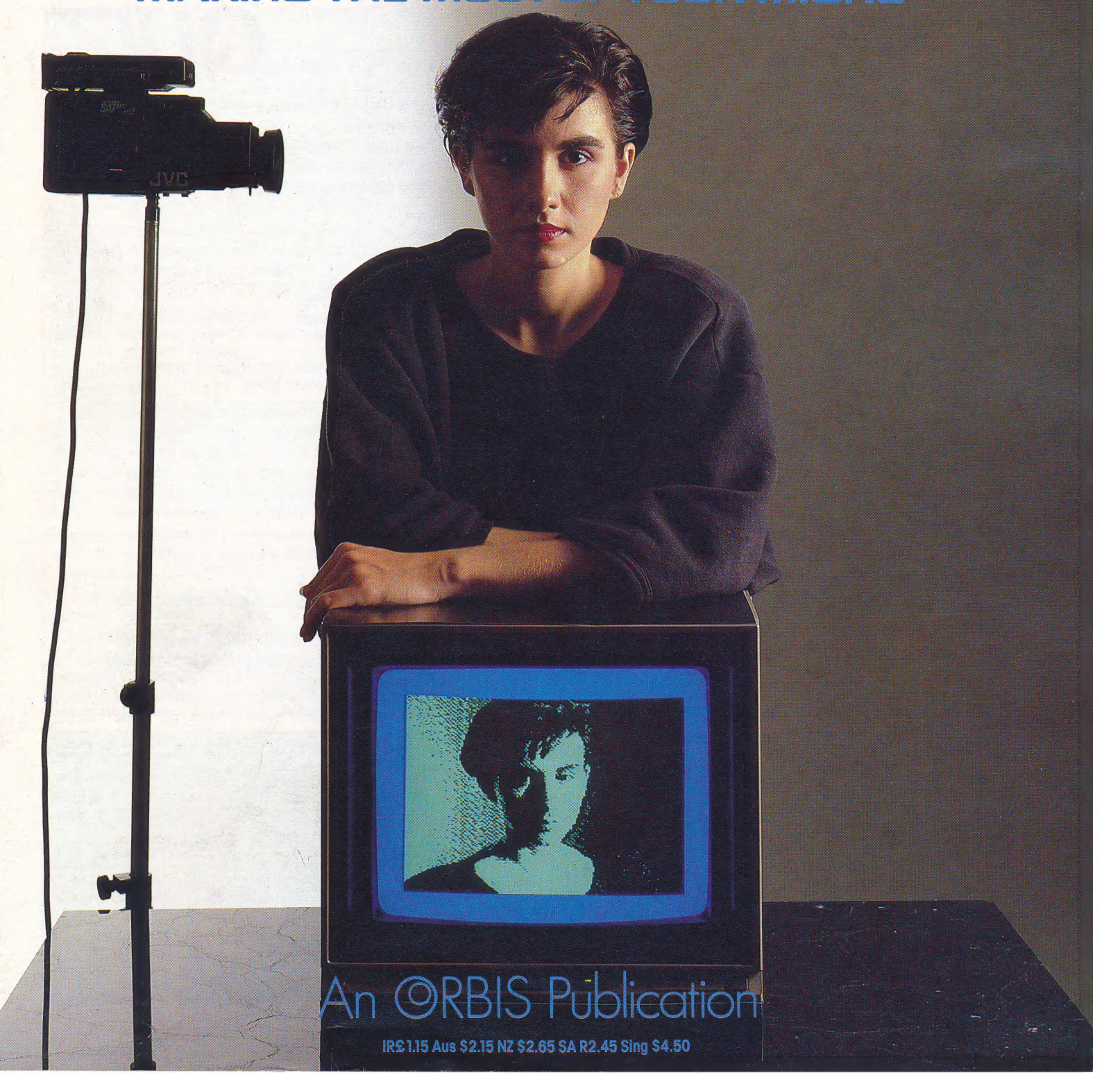
HUMPHRIES

ISSN 0265-2919

90p 83

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO



An ORBIS Publication

IR£1.15 Aus $2.15 NZ $2.65 SA R2.45 Sing $4.50

CONTENTS

APPLICATION

PROBABLY A GOOD BET Scientific approaches to gambling could increase your chances of winning 1649

HARDWARE

ALARMING IMAGES The Print-Technik Video Digitiser for the Commodore 64 1641

SOFTWARE

SNEAK PREVIEW We prepare for the final listings with a look at methods of sorting 1646

COMPUTER SCIENCE

NEW TRICKS FOR AN OLD LANGUAGE FORTRAN has undergone a number of changes since it was first introduced in the late 1950s 1654

JARGON

FROM SOURCE CODE TO STACK A weekly glossary of computing terms 1660

PROGRAMMING PROJECTS

CONFINED TO CELLS We begin to enter and manipulate data in our spreadsheet 1652

MACHINE CODE

DESIRABLE RESIDENTS Our Amstrad OS series continues with a look at resident system extensions 1657

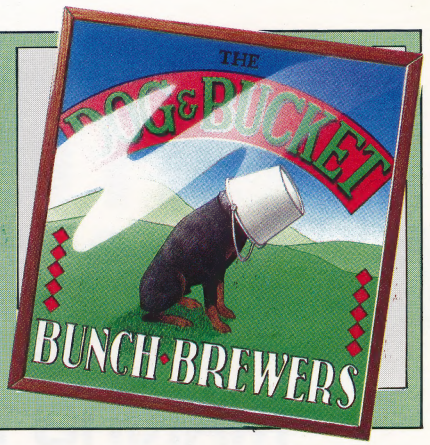
WORKSHOP

MIDI IN We conclude our adaptation of the MIDI interface for the Amstrad 1643

WHEELS Two road racing packages INSIDE BACK COVER

Next Week

- **In the final instalment of our series on the Amstrad operating system, we examine the keyboard manager.**
- **The Dog and Bucket adventure game reaches its climax as we present the listing in its entirety.**
- **Digital Research's GEM environment is highlighted in the first of a short series on this WIMP-based operating system.**



QUIZ

1) In FORTRAN, what is the purpose of a 'preprocessor'?
2) How long does it take for the Print-Technik Video Digitiser to digitise an image?
3) What does the term 'polling' refer to?

Answers To Last Week's Quiz
1) SOUND_HOLD suspends any envelopes that are in use and disables the sound event — thus halting all sound generation.
2) 61.5 Kbytes.
3) Synchronisation. Often the slaves will be idle, awaiting further instructions from the master machine.

Coming Up . . .

- A programming project series on methods of text compression.
- A series of reviews on wordprocessing software.
- A look at Unix, the possible successor to MS-DOS as the industry standard.

Editor Stephen Cooke; **Art Editor** Claudia Zeff; **Deputy Editor** Steve Colwill; **Production Editors** Bobby Pickering, Jon Kaye; **Designers** Julian Dorr, Mike Clowes; **Staff Writer** Steve Malone; **Art Assistant** Caroline Clayton; **Sub Editor** Nik Lumsden; **Contributors** Jon Kaye, Chris Honey, Chris Laing, Dougie Bern, Tony Drapkin, Pete Connor, Mike Curtis, Steve Cooke, Steve Colwill, Steve Malone, Martin Young; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd; Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Assistant** Susan Brown; **Subscription Manager** Christine Allen; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; © **APSIF Copenhagen 1985; © Orbis Publishing Ltd 1985: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Heanor Gate Printing Ltd, Derby**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA –** Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders.
UK/EIRE – Issue Price: 90p/IR£1.15. Subscription: 6 months: £26.00. 1 Year: £52.00. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Issue Price: 90p. Subscription: 6 months air: £44.72. Surface: £36.14. 1 year air: £89.44. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Issue Price: 90p. Subscription: 6 months air: £50.18. Surface: £36.14. 1 year air: £100.36. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **AMERICAS/ASIA/AFRICA** – Issue Price: US/CAN$1.95/90p. Subscription: 6 months air: £59.54. Surface: £36.14. 1 year air: £119.08. Surface: £72.28. Binder: £5.00. Airmail: £9.50. **SOUTH AFRICA** – Issue Price: SA R2.45. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Issue Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Issue Price: 90p. Subscription: 6 months air: £64.22. Surface: £36.14. 1 year air £128.44. Surface: £72.28. Binder: £5.00. Airmail: £9.75. **AUSTRALIA** – Issue Price: AusS2.15. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Issue Price: NZS2.65. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 5211. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited. Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER PHOTOGRAPHY BY MARCUS WILSON-SMITH VIDEO CAMERA COURTESY OF UNIQUE SOUNDS LTD.

ALARMING IMAGES

The technology that converts a video signal into a form that a computer can process is becoming very compact and reasonably affordable. The Austrian-made Print-Technik Video Digitiser for the Commodore 64 lets us sample this technology for ourselves.

A video digitiser takes a signal from any video source (usually a camera but perhaps a laser disc or video recorder) and converts it from the analogue voltages of the signal into a large set of numbers representing the image. These numbers can be stored in a computer's memory, displayed in high-resolution graphics, manipulated by software, dumped to a printer and so on.

There are hundreds of possible uses for video digitisers, ranging from serious applications, such as alarm systems, to amusing ideas such as 'computer pictures' and badges of people. Print-Technik cites the example of a hairdresser who digitises images of his customers. By using a light pen and graphics package, he is able to show the customer a variety of hair styles to help choose a suitable coiffure.

The Print-Technik unit, however, doesn't look as if it is capable of all this. It's a tiny black box that slots into the Commodore 64's user port. A standard video socket on this connects to your video source — there are no other leads as the unit takes its power from the computer. The only other 'controls' are three tiny adjustable screws, which alter the brightness, contrast and width of the digitised image.

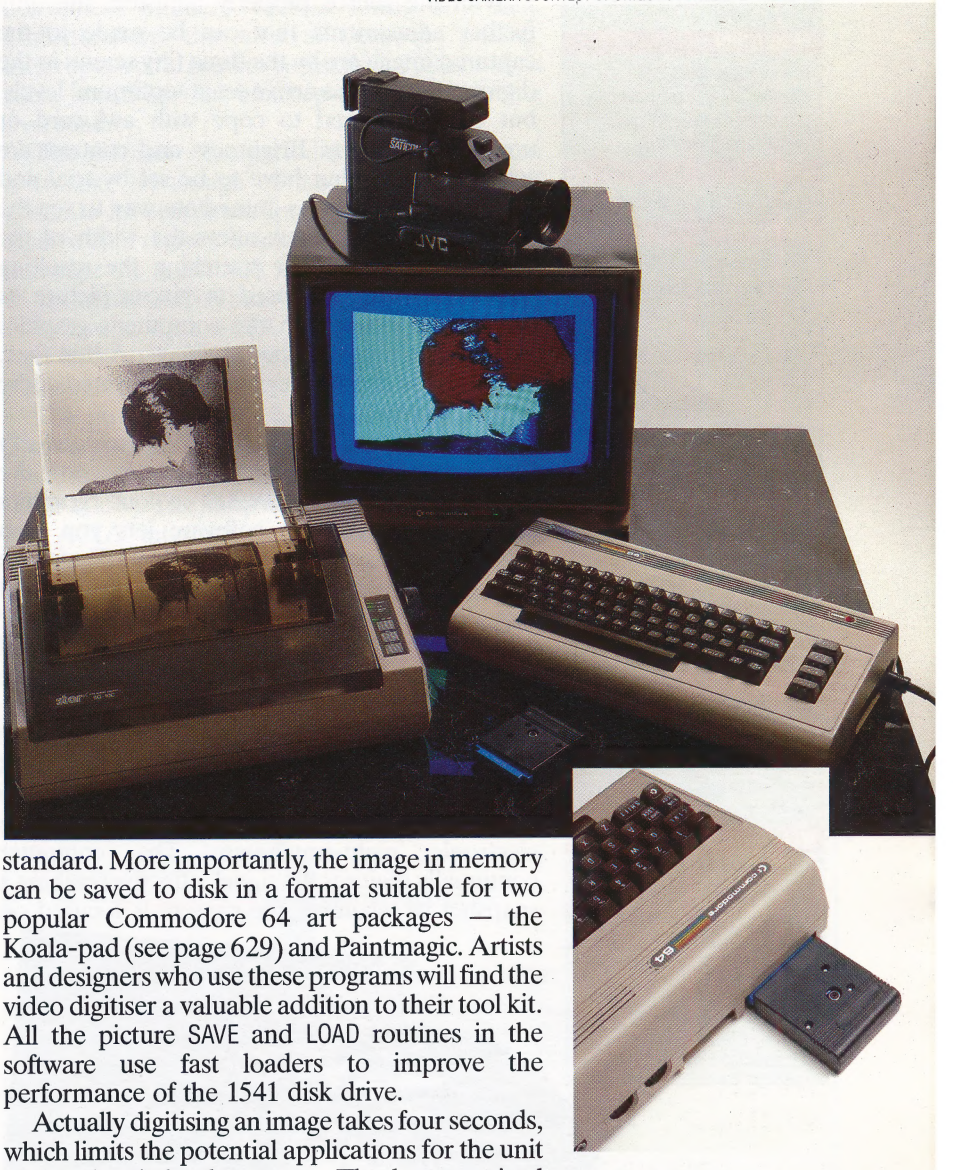
The unit is controlled through a software package supplied on disk. This is a well-constructed and functional program, but it does little to exploit the true possibilities of the unit. The program presents a friendly menu from which options are selected by moving a hand-shaped pointer with the cursor keys and pressing the return key. You can opt to digitise the signal, view the picture currently in memory, save it to disk or print it. A variety of printers are supported, including Commodore's 801 and 1525 units. A special printer program called '16 colors' can be loaded to print pictures in up to 16 colours using a choice of the five leading colour dot-matrix printers. On-screen pictures are limited to four colours but the digitiser itself is capable of resolving up to 256 shades.

Sensibly, the software is designed to be used in conjunction with other programs. A Lightpen option calls in Print-Technik's own light pen graphics software, although this isn't supplied as standard. More importantly, the image in memory can be saved to disk in a format suitable for two popular Commodore 64 art packages — the Koala-pad (see page 629) and Paintmagic. Artists and designers who use these programs will find the video digitiser a valuable addition to their tool kit. All the picture SAVE and LOAD routines in the software use fast loaders to improve the performance of the 1541 disk drive.

Actually digitising an image takes four seconds, which limits the potential applications for the unit and makes it harder to use. The lowest priced system you could set up would be based around a closed-circuit television camera similar to those used for surveillance purposes. These cameras have no internal monitor/viewfinder. It's therefore almost mandatory to have a normal video monitor available so that the subject can be aligned and the camera focused visually by the operator before the camera is disconnected from the monitor and connected to the digitiser, ready to capture the image. So although the system works without an external monitor, it becomes a trial and error affair, lessening to a large extent, the quality of the results.

The four seconds taken to produce an image also make it difficult to capture a moving subject.

VIDEO CAMERA COURTESY OF UNIQUE SOUNDS LTD. MARCUS WILSON-SMITH



Video Techniques
Print Technik's video digitiser for the Commodore 64 takes video images from a video camera or recorder and digitises them, allowing them to be displayed on the screen in four shades, stored on disk or printed out. A particulary useful facility allows the digitised images to be saved on disk in formats used by other graphics packages such as the Koala-pad. The package includes a small cartridge that plugs into the user port and several good-quality programs on disk to support the hardware

PRINT-TECHNIK VIDEO DIGITISER

PRICE
£149.95 (inc VAT)

DIMENSIONS
80×65×20mm

EXTERNAL CONTROLS
Brightness, contrast and picture width

RESOLUTION
256×256 pixels in 256 shades. Images are normally presented in 160×200 pixel format using four shades

SPEED
Four seconds per image

SOFTWARE
Standard software allows images to be digitised, saved in 256×256, Koala-pad or Paintmagic format, printed and re-coloured. Additionally, demonstration programs provide an alarm system and slide show, and allow you to use digitised pictures in your own BASIC programs

DOCUMENTATION
A poorly translated four-page booklet fails to cover the more technical aspects of the unit. However, basic operation is simple enough for this not to be a problem

STRENGTHS
The unit is compact, reasonably priced and works extremely well. The software supplied is adequate for most simple applications

WEAKNESSES
Slow response means that the unit should be used with an external monitor. The limits of the Commodore 64's graphics screen (four colours at 160×200 pixels) means that the capabilities provided are not fully exploited in the standard package

Quality pictures of people involve the subject sitting still for the entire four seconds, although some bizarre effects can be created by movement while the digitiser is working. There are two solutions to this problem. The ideal one is to use a video 'frame grabber', which will freeze the action electronically. The alternative is simply to take a photograph of the subject and digitise that!

With the source properly adjusted, the only further adjustments that can be made to the captured image are by the three tiny screws in the digitiser unit. These arrive set at 'optimum' levels, but can be altered to cope with awkward or unusual conditions. Brightness and contrast are self-explanatory but have to be set by trial and error, since there is no immediate way to see the results. The third screw alters the width of the image, compressing or stretching the resulting picture. This can be used to give a picture its correct proportions on the computer's graphics screen or to produce special effects. A thin face, for example, can be magically transformed at the turn of a screw.

The digitised image is 256×256 pixels and can therefore only be partially seen on the Commodore 64's 160×200 colour graphics screen. Print-Technik's software lets you scan around the image using the cursor keys. The image is normally presented in four shades — white, light grey, dark grey and black — although the digitiser itself works with 256 shades. You can select any of four colours with the function keys using the supplied program.

A number of alternative sample programs are also supplied, of which Alarm is the most sophisticated. This allows you to use your video camera in conjunction with the digitiser as an electronic night-watchman. The computer continually digitises the signal, effectively taking a snapshot of whatever the camera is focused on around every five seconds.

When the new image is ready, it's compared to the image recorded five seconds before and if it is different in more than a set number of places, the Commodore 64 dutifully sounds an alarm. This program worked remarkably well with the system being used to keep watch on a coffee cup. When the number of differences that trigger the alarm is low (around 200) the alarm will sound even if someone just casts a light shadow over the watched area. Obviously, if it was guarding some prized jewels, the difference limit would be set higher so that the alarm did not trigger at every patron's shadow or when daylight faded.

This program gives a very favourable impression of the sensitivity of Print-Technik's unit. With a Commodore 64 wired to a real alarm system, a thief would have a one in five chance of remaining undetected — providing that the scene could be exactly restored within one second!

Print-Technik also includes a simple slide show program that allows pictures saved in Koala-pad format to be shown in succession on the screen. Finally, there's a routine that allows you to display Koala-pad format pictures within your own BASIC programs using a SYS call.

Print-Technik's digitiser transforms the Commodore 64 into a tool for producing high-quality graphic images. The hardware is compact, produces surprisingly good results, and the supplied software is comprehensive although it is not as well finished as some programs. However, any practical application of the unit requires other equipment — a video camera, video monitor and a reasonable art program such as Paintmagic or the Koala-pad. The unit's appeal is therefore limited to those who have a serious use for it as a tool for producing interesting and detailed graphics or using some of the more interactive applications described here.

Print Possibilities
Digisted images can be dumped to a range of printers. Commodore's MPS 801 and 1525 printers each produce a four-shade monochrome image on paper but the picture can be printed in up to 16 colours with a colour printer

MIDI IN

We complete our MIDI interface for the Amstrad CPC range by developing a piece of machine code software to run the interface under Amstrad's firmware interrupt system. We'll also be demonstrating the principles of buffering incoming data from an asynchronously linked device.

In our initial MIDI project for the Commodore 64 and BBC Micros, we developed a program that allowed a computer to act as a simple real-time recorder. Because this program (see page 1412) was simplified to demonstrate the principles of MIDI, incoming data was subject to a minimum amount of processing before storage. This processing took the form of checking for system messages, discarding any encountered (see page 1387), and calculating the timing bytes to be inserted between the MIDI bytes before storing the data in memory. Since the minimum time interval between two consecutive MIDI bytes is 320µs, the processing loop for each incoming byte was kept much shorter than this critical time period, to ensure that all processing was done and that the program was in a state to receive the next byte before it arrived.

A more useful program would have to carry out much more extensive processing of the incoming data. Features such as simultaneous record and playback, screen display and multi-channel overdubbing could be achieved if it were not for the 320µs time limit on processing. Fortunately, there is a way, commonly implemented in computer systems, of 'listening' for incoming bytes while simultaneously processing data that has already been received.

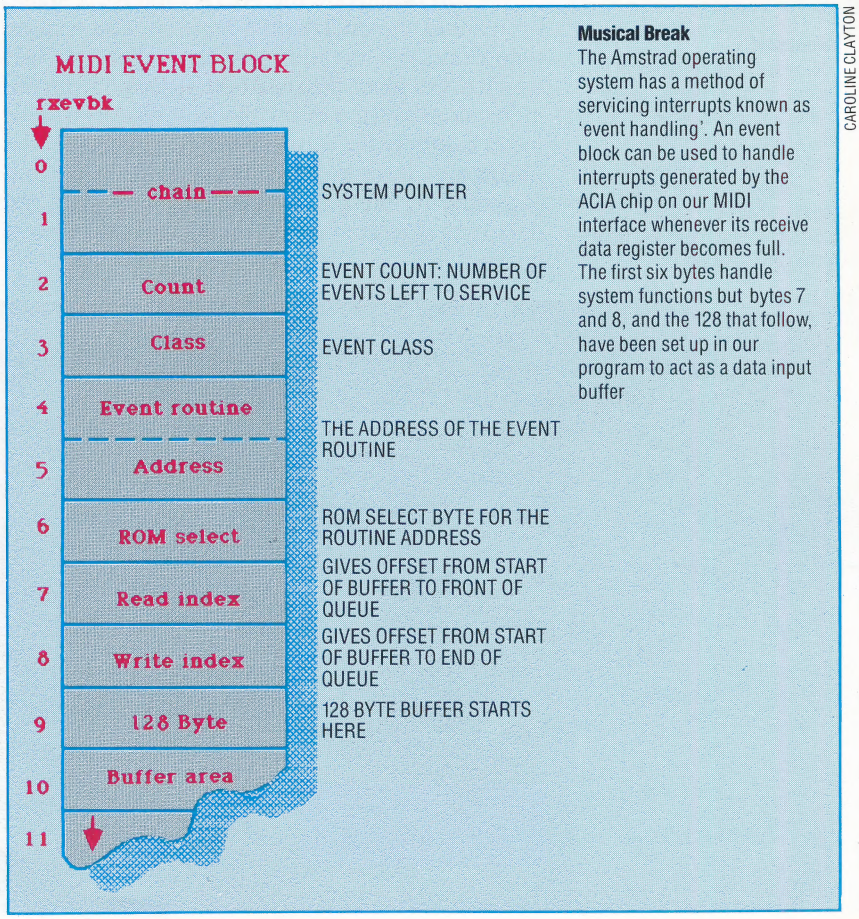
To demonstrate the principles of writing a MIDI program that has long processing routines (longer than 320µs) let's look at a program to read incoming MIDI data and display it on the screen in real time. The data will be displayed in hexadecimal notation with each MIDI message starting on a new line. To do this we need to convert each MIDI byte received into two hexadecimal digits in ASCII code, send them to the screen followed by two space characters to separate them from the next byte and, if the byte is the final one in the MIDI message, send a carriage return. Operating system overheads, such as keyboard scanning, are significant in the context of a 320µs data transmission rate and so it is likely that such a processing routine would take longer than the critical period between reception of each successive MIDI byte.

BUFFERING INPUT DATA

So that we don't lose data as it comes in, we could use a regular interrupt to our main processing loop that scans the ACIA status register and transfers any newly received data from the ACIA receive data register to a buffer area in the computer's memory. The foreground program could then read data from the front of the buffer at a convenient moment without losing any data. Unfortunately, the interval between 'polling' interrupts would need to be less than 320µs and, as such, constitutes an unacceptable programming overhead.

The alternative to a periodic interrupt structure is to take advantage of the ACIA's ability to generate its own interrupt signal every time the receive data register is full. Of course, we need to intercept the Amstrad's interrupt service routine to distinguish ACIA interrupts from other sources of interrupt. If an ACIA interrupt was detected, we'd have to transfer the contents of the receive data register.

The program shown here uses the Amstrad's firmware routines, which are fully documented in the firmware manual and available from Amstrad (SOFT 158). The jump to the routine for handling the ACIA interrupt is patched into location &003B. The interrupt-handling routine itself communicates with the 'foreground' processing program by 'kicking' (see page 1598) an event, which is described by the event block located at the address labelled rxevbk.



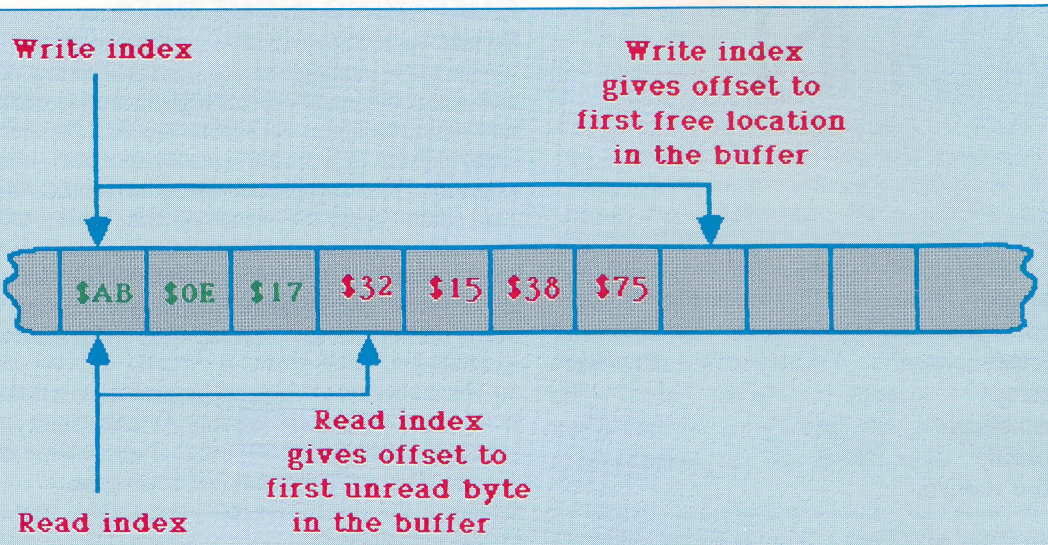
Musical Break

The Amstrad operating system has a method of servicing interrupts known as 'event handling'. An event block can be used to handle interrupts generated by the ACIA chip on our MIDI interface whenever its receive data register becomes full. The first six bytes handle system functions but bytes 7 and 8, and the 128 that follow, have been set up in our program to act as a data input buffer

CAROLINE CLAYTON

Buffer Pointers

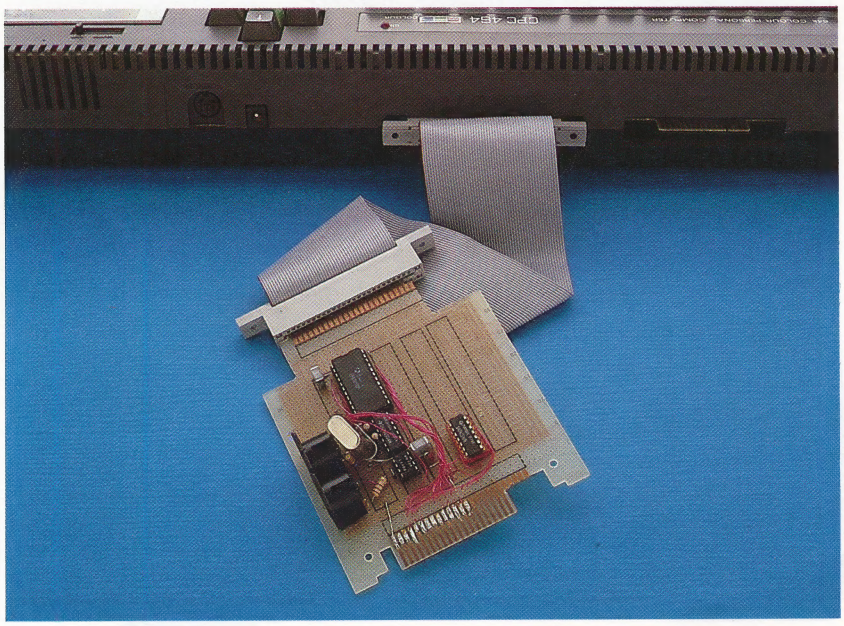
Pointing The Way
The pointers to the front and end of the buffer queue indicate the points at which data is to be removed or inserted. During operation, the pointers will move progressively through the buffer area as data is removed and new data is added. Whenever the buffer becomes empty (indicated by the READ and WRITE indices having the same value) the indices are reset to the beginning of the buffer



The buffer area is defined as 128 bytes of memory starting at address rxevbk +9. The routine first checks to see if there are any outstanding events waiting to be processed. If not, the buffer is empty and the read and write indices are initialised. The routine then goes on to check for receiver overrun — another byte arriving at the ACIA before the previous one had been read. The interrupt source must be cleared by the interrupt routine or it will repeat itself each time the handler routine is exited and interrupts re-enabled, causing the machine to 'lock'.

We can deactivate the ACIA's interrupt signal by reading the ACIA data register. The routine then works out the address in the buffer area into which it will place the data byte received from the index stored in the event block at rxevbk +8. This index is simply the offset of the current end of the buffer area from its head. If the buffer is not full (signified by the index being less than 128), the data byte is written to the buffer and the event is 'kicked'.

Interfaced
Our redesigned MIDI board connects via a 50-way ribbon cable and edge connectors to the Amstrad's expansion port. Using standard 5-pin DIN connectors, up to 16 MIDI-equipped devices can be connected and controlled from computer software



The 'foreground' program simply initialises the ACIA registers and the event block, after which it enters a loop that takes characters from the front of the buffer and processes them in the manner described previously. The address of the location representing the front of the buffer is found from the read index, at address rxevbk +7 within the event block. This index is again held as an offset from the start of the buffer.

The program exits if any key on the computer keyboard is pressed, if the buffer becomes full or if the program misses any data bytes because of receiver overrun.

PROGRAM PROBLEMS

The program given here is designed to demonstrate some of the principles of using non-periodic interrupts with the Amstrad operating system. It's interesting to note some of the difficulties of using such a method. If you run the program and send data from a MIDI keyboard, you will find that the program will occasionally end because of an ACIA overrun — a second byte is received before the previous byte is cleared from the receive data register. This happens because of the way in which the OS handles interrupts.

The Amstrad's interrupt response uses the Z80's alternate register temporarily set to store the state of the CPU registers whenever an interrupt occurs. Since only one set of alternate registers is available, multiple level interrupts are not supported. This in turn means that interrupts must be disabled throughout the interrupt service routine. Additionally, because the service routine includes several calls to support the firmware's event structure, interrupts may be disabled for longer than the crucial 320µs minimum period, possibly causing one or more bytes to be lost.

A proper solution to this overrun problem would involve replacing the interrupt handling code with our own streamlined code. This would effectively ignore periodic interrupts and seriously limit the use we could make of the firmware's system functions.

Hex Loader:

```
20 org=&408E:loadptr=0
30 WHILE flag=0
40 GOSUB 1000:IF flag=1 THEN 60:REM exit
50 POKE org+loadptr,entry:loadptr=loadptr+1
60 WEND
70 END
1000 REM **** get a hex byte s/r ****
1020 flag=0  LINE INPUT entry$
1025 IF entry$="x" THEN flag=1:RETURN
1030 IF LEN(entry$)<>2 THEN PRINT"error"
:GOTO 1020
1050 entry=VAL("&"+entry$):RETURN
```

If you don't have an assembler, the machine code program can still be loaded using the BASIC hex loader given here. Bytes from the left-hand column should be entered one at a time with the hex loader running, and then saved to disk or tape using the command SAVE 'FILENAME', B,&4000,&199. Note that each byte must be followed by a carriage return; so, for example, the three bytes on the second line, 113B00, should be intered as 11 <CR> 3B <CR> 00 <CR>

Amstrad MIDI Program

```
                org  #4000
        klev:   defs 2
        rxevbk: defs 140          ;event block + read/write
        cont:   equ  #f8e0
        txreg:  equ  #f9e0
        stat:   equ  #fae0
        rxreg:  equ  #fbe0
        ;os call addresses
        kmrdch: equ  #bb09
        kmarbr: equ  #bb45        ;arm breaks
        txtout: equ  #bb5a
        klinev: equ  #bcef
        kleven: equ  #bcf2
        kldisa: equ  #bd0a        ;disarm async event
        klpoll: equ  #b921        ;carry returns event
        klnext: equ  #bcfb        ;get next sync event
        kldosy: equ  #bcfe        ;do sync event
        kldone: equ  #bd01        ;done sync event
        klsres: equ  #bcf5        ;clear event queue
        start:                    ;patch interrupt routine

F3              di
113B00          ld   de,#003b
211441          ld   hl,jpinst
010300          ld   bc,3
EDB0            ldir
FB              ei
        ;initialise receive event
210240          ld   hl,rxevbk
115941          ld   de,rxevrt    ;event routine address
0601            ld   b,1
CDEFBC          call klinev
3600            ld   (hl),0       ;initialise read/write
23              inc  hl
3600            ld   (hl),0
        ;save kl event routine
        addr
ED5BF3BC        ld   de,(kleven+1)
CBBA            res  7,d
CBB2            res  6,d          ;remove rom select bits
ED530040        ld   (klev),de
        ;initialise 6850
01E0F8          ld   bc,cont
3E03            ld   a,3
ED79            out  (c),a
3E96            ld   a,#96
ED79            out  (c),a
        wait:
CDFBBC          call klnext
DC0941          call c,dosync     ;do event if waiting
210A40          ld   hl,rxevbk+8
CB7E            bit  7,(hl)
2005            jr   nz,wait1     ;exit if buffer full
CD09BB          call kmrdch
30EE            jr   nc,wait      ;jump if no key pressed
        wait1:
CB76            bit  6,(hl)
280C            jr   z,wait2      ;skip message if not over
21F340          ld   hl,string
0616            ld   b,strlen
7E      nexc:   ld   a,(hl)
CD5ABB          call txtout
23              inc  hl
10F9            djnz nexc
        wait2:
CDF5BC          call klsres

F3              di
3EC9            ld   a,#c9        ;ret instruction
323B00          ld   (#003b),a
3E16            ld   a,#16
ED79            out  (c),a        ;disable 6850 interrupts
FB              ei
C9              ret               ;exit to editor
0A0D    string: defb #0a,#0d
41434941        defm "ACIA
                overrun error"
0A0D            defb #0a,#0d
        strlen: equ  $-string
        dosync:                   ;perform sync event
E5              push hl           ;save event address
F5              push af           ;save previous priority
CDFEBC          call kldosy       ;perform event
F1              pop  af
E1              pop  hl
CD01BD          call kldone
C9              ret
C31741  jpinst: jp   rcvint
        rcvint:                   ;rcv interrupt routine
3A0440          ld   a,(rxevbk+2)
B7              or   a            ;test for count = zero
2006            jr   nz,rcv1
320940          ld   (rxevbk+7),a ;initialise indices
320A40          ld   (rxevbk+8),a
C5      rcv1:   push bc           ;save old rom state
210A40          ld   hl,rxevbk+8  ;write index address
CB91            res  2,c          ;enable lower rom
ED49            out  (c),c
01E0FA          ld   bc,stat
ED78            in   a,(c)
CB6F            bit  5,a
280A            jr   z,rcv15
36FF            ld   (hl),#ff     ;set overrun flag
3E16            ld   a,#16
05              dec  b
05              dec  b
ED79            out  (c),a
1817            jr   rcv2
04      rcv15:  inc  b
ED78            in   a,(c)
34              inc  (hl)         ;increment index
5E              ld   e,(hl)
CB7B            bit  7,e          ;check for buffer full
200E            jr   nz,rcv2      ;exit if full
1600            ld   d,0
19              add  hl,de        ;get write address
77              ld   (hl),a
210240          ld   hl,rxevbk
ED5B0040        ld   de,(klev)
CD1600          call #0016
C1      rcv2:   pop  bc
ED49            out  (c),c
C9              ret
        rxevrt:                   ;rx event routine hl=even
210940          ld   hl,rxevbk+7
34              inc  (hl)         ;increment index
5E              ld   e,(hl)
1C              inc  e
1600            ld   d,0
19              add  hl,de        ;point to next read data
7E              ld   a,(hl)
FEF8            cp   #f8
D0              ret  nc
B7              or   a
F27641          jp   p,evrt1
47              ld   b,a
3E0A            ld   a,#0a
CD5ABB          call txtout
3E0D            ld   a,#0d
CD5ABB          call txtout
78              ld   a,b
        evrt1:
CD7A41          call hexpr        ;print char
C9              ret
        hexpr:                    ;hexadecimal print
0602            ld   b,2
        hexpr1:
3E00            ld   a,0
ED6F            rld               ;get ms digit
CD9141          call digasc       ;convert to ascii
CD5ABB          call txtout       ;print
10F4            djnz hexpr1
3E20            ld   a," "
CD5ABB          call txtout
CD5ABB          call txtout
C9              ret
        digasc:                   ;hex  digit to ascii
C630            add  a,"0"
FE3A            cp   ":"
D8              ret  c
C607            add  a,"A"-":"
C9              ret
```

SNEAK PREVIEW

In this instalment, we discuss the implications of programming tree structures of differing types. We add some sorting routines to our program to enable it to handle all trees, regardless of their structure.

Using trees with irregular internal structures — for example, three branches from one node, two from another, four from a third, and so on — poses special problems. The main difficulty is knowing whether or not we have reached a terminal node. As you may recall, when we entered the object manipulation tree, we were able, because of its internal consistency, to number the nodes in a special order: choice nodes first; nodes that jumped out of the tree and then back in second; then terminal nodes that resulted in an action or message; and finally, terminal nodes that simply returned without any action being taken. When we sorted through the tree (in lines 5030-5090) we simply jumped to a routine as determined by the number of the node.

Unfortunately, if you look at the trees on pages 1624 and 1625 and attempt to apply this system to them, you will find it just isn't possible.

What's needed is a foolproof system of sorting a tree that can be applied to *any* tree, regardless of its internal structure.

We shall now solve this problem, and enter our plot tree into the program. Here's how it's done . . .

First, we need to initialise the variables for the plot which have not already been used. Delete line 190 of your program and add the following line:

```
190 DIM t(5,25,4),k(3,30),c(25),s(6),h(6
): z=0
```

You will note that we have enlarged the t array to cope with our new trees, and have also added some space into the c array for the new conditions that we shall be testing. The two arrays s and h record respectively whether or not a character has seen a corpse or handled the empty cat food tin. The variable z indicates whether a death has taken place, and will be set when appropriate to indicate the character number of the victim.

We will program our tree as follows. The four bottom-dimension elements of the t array will be used to hold information about each node in the tree as indicated in the Node Type table. It will then be a simple matter, when sorting through the tree, to check the node type of each and jump to the appropriate routine. The node data is stored in Listing 1, which you should enter first. Then enter the following two lines to read this data into the t array:

```
230 REM plot tree
240 FOR n=1 TO 22: FOR s=1 TO 4: READ t(
2,n,s): NEXT s: READ a$: NEXT n
```

Note the blank spaces in lines 6270 and 6280. These are added to aid readability, so that you can see each group of node values at a glance. The blank spaces are read into the 'garbage variable' a$ in line 240.

We now need a routine to 'sort' this tree, and jump to the relevant routines. this is held in lines 5400 to 5550 (Listing 2) which you should now enter. The process is very straightforward. Line 5470 checks the node type and adds one to it to generate a number between 1 and 7 which then results in a jump to one of the lines indicated. A simple choice node with only two branches (type 0) will jump to 5480, where the value of the condition held in t(2, node number, 2) is used to decide whether to jump to the node held in t(2, node number, 3) or t(2, node number,4).

The remaining node types each jump to their own lines and have the relevant parameters extracted from the t array. Note that node types 1 and 2 are vectored via a jumpblock held in lines 4500-4570. This is to avoid having to clutter up the tree-sorting routine with lots of different line numbers as we add more trees to the program.

If you refer to the plot tree diagram, you will see that a number of new conditions need to be tested that we have not already stored in the c array. Enter the following line to add these in:

```
2450 c(13)=ABS(z=c): c(14)=ABS(g=c): c(1
5)=ABS(z=0): c(16)=ABS(s(c)=255): c(17)=
ABS(FNc(z,2)=FNc(c,2)): c(18)=ABS(h(c)=2
55): c(19)=ABS(i=2)
```

Now enter Listings 3, 4, and 5. We are now almost ready to test this part of the program. All that's required is first to alter the control loop in the line so that it reads as follows:

```
500 REM
510 REM test program loop
520 REM
530 GOSUB 2100: GOSUB 2150: GOSUB 2240:
PRINT: PRINT: GOSUB 1000: GOTO 530
```

Now delete lines 540 to 820, which we entered when testing our object manipulation tree, and

Calculated Chance

Node	Type	Data held in t(tree number,node number,1-4) 1	2	3	4
0	Choice node	0	condition no.	node to jump to if FALSE	TRUE
1	GOSUB nodes (exit tree and then return)	1	0	node to return to	routine no.
2	Action node (terminal node, jumps to routine)	2	0	routine no.	message no.-if any
3	Message node (terminal node, prints message)	3	0	0	message no.
4	Terminal node (exit tree, take no action)	4	0	0	0
5	Random node (generate random offset and add to base node)	5	0	base node	maximum offset
6	Multiple choice (add value of condition to base node)	6	condition	base node	0

type in Listing 6. This listing checks the handle and move flags, initialises conditions, and calls each tree in turn in lines 1200 and 1210. You can now run the program. At this stage, it is possible for characters to move about, manipulate objects, die, and even solve the mystery of the poisonous pasties.

In the next instalment we print the first part of the complete listing, which will add the final touches and enable you to enter the full program.

Listing One

These lines hold the data for the nodes. The dummy spaces have been included to aid readability.

```
6240 REM
6250 REM plot tree data
6260 REM
6270 DATA 0,13,2,22," ",0,14,5,3," ",0,1
5,21,4," ",5,0,18,3," ",0,16,6,7," ",0,1
7,7,11," ",0,18,9,8," ",0,17,12,13," ",0
,19,10,17," ",5,0,14,3," ",1,0,7,1," ",4
,0,0,0," ",2,0,1,0," ",2,0,5,1," ",4,0,0
,0," ",4,0,0,0," ",2,0,2,4," ",2,0,3,2,"
 ",2,0,4,3
6280 DATA " ",2,0,4,0," ",4,0,0,0," ",4,
0,0,0," "
```

Listing Two

These lines sort through the trees, testing the type of each node and branching accordingly.

```
5440 REM
5450 REM sort trees
5460 n=1
5470 ON (t(t,n,1)+1) GOTO 5480,5490,5500
,5520,5530,5540,5550
5480 k=c(t(t,n,2))+1: n=t(t,n,2+k): GOTO
 5470
5490 GOSUB 4530 : n=t(t,n,3): GOTO 5470
5500 GOSUB 4570
5510 RETURN
5520 GOSUB 4680: GOSUB 4630: GOSUB 4720
5530 RETURN
5540 a=t(t,n,4): GOSUB 4350: n=t(t,n,3)+
v: GOTO 5470
5550 k=c(t(t,n,2)): n=t(t,n,3)+k: GOTO 5
470
```

Action Stations

Lines 2810-2920 take care of moving characters from room to room. Lines 3000-3110 from the first part of the action table. This table contains the routines called by type 2 nodes via the jumpblock in lines 4510-4570. Routines for the 'GOSUB nodes' (type 1) begin at line 3900.

Listing Three

```
2810 REM
2820 REM move a character
2830 REM
2840 IF FNc(c,4)<1 THEN RETURN: REM too
weak to move
2850 y=0: f=0: FOR w=2 TO 5: IF 1$(VAL(c
$(c,2)),w)="0" THEN GOTO 2910
2860 GOSUB 4180: IF q=1 THEN f=1: GOTO 2
880
2870 GOTO 2910
2880 IF FNc(c,2)=r THEN PRINT c$(c,1);"
leaves the room...";: y=1
2890 c$(c,2)=1$(VAL(c$(c,2)),w): w=5: IF
 FNc(c,2)=r THEN PRINT c$(c,1);" enters
the room...";: y=1
2900 IF y=1 THEN y=c: GOSUB 2250: c=y: P
RINT: PRINT: REM update characters prese
nt message
2910 NEXT w: IF f=0 GOTO 2850
2920 RETURN
3000 REM
3010 REM action table
3020 REM
```

Useful Routines

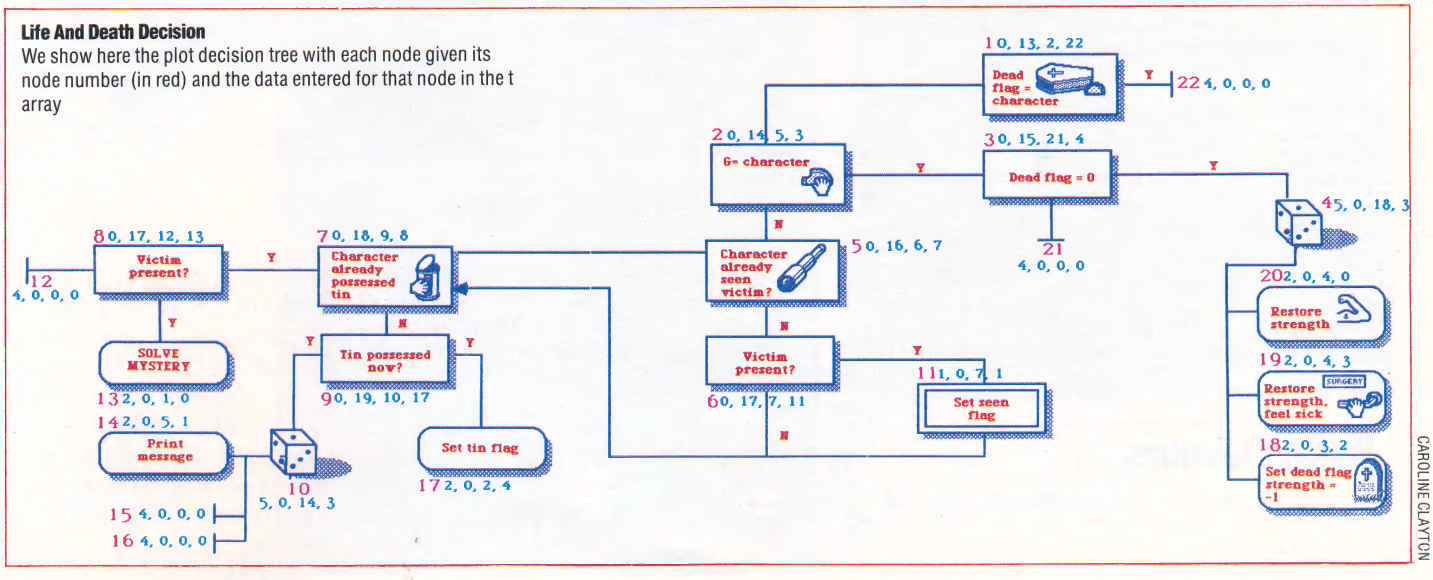
This part of the program adds some new low-level routines, for generating variable random numbers and printing messages

Listing Four

```
4270 REM
4280 REM print his/her
4290 REM
4300 IF c$(c,7)="f" THEN m$="her ": RETU
RN
4310 m$="his ": RETURN
4320 REM
4330 REM variable random number routine
4340 REM
4350 v=INT(RND(2)*a): RETURN
4360 REM
4370 REM print he/she
4380 REM
4390 IF c$(c,7)="f" THEN PRINT "she ";:
RETURN
4400 PRINT "he ";: RETURN
4500 REM
4510 REM jumpblock
4520 REM
4530 REM re-entrant routine
4540 ON t(t,n,4) GOSUB 39[illegible]
4550 RETURN
```

Life And Death Decision

We show here the plot decision tree with each node given its node number (in red) and the data entered for that node in the t array



CAROLINE CLAYTON

```
3030 FOR n=1 TO 3: GOSUB 4090: NEXT n: m
$=c$(c,1)+" takes another look at the bo
dy, and suddenly realises the hideous tr
uth. "+CHR$(34)+"The pasty's made of cat
-food"+CHR$(34)+" ": GOSUB 4630
3040 GOSUB 4390: m$=m$+"yells, and in a
mad rush the assembled company scramble
over the bar and attack Fred the Barman,
 who pleads with them to no avail to spa
re his miserable life.": GOSUB 4630: GOS
UB 4720
3050 FOR n=1 TO 2000: NEXT n: GOSUB 4720
: m$="...and the next day Fred the Barma
n is nowhere to be seen.  However, a num
ber  of oddly shaped pasties are availab
le to feed the ever-hungry clientele of
the Dog and Bucket...": GOSUB 4630: GOSU
B 4720: END
3060 h(c)=255: GOSUB 4680: m$=c$(c,1)+m$
: GOSUB 4630: GOSUB 4720: RETURN
3070 z=c: GOSUB 4680: n$=c$(c,1)+m$: GOS
UB 4300: m$=n$+m$+"stomach, and immediat
ely expires.  The other characters are t
oo involved with their drinks to notice.
..": GOSUB 4630: GOSUB 4720: g=0: c$(c,4
)="-1": RETURN
3080 c$(c,4)="10": g=0: IF t(t,n,4)>0 TH
EN GOSUB 4680: m$=c$(c,1)+m$: GOSUB 4630
: GOSUB 4720
3090 RETURN
3100 a=20: GOSUB 4350: IF v<>5 THEN RETURN
3110 GOSUB 4680: GOSUB 4630: GOSUB 4720:
 RETURN
3900 REM
3910 REM gosub table
3920 REM
3930 s(c)=255: m$=c$(c,1)+" kneels down
beside the prostrate body of "+c$(z,1)+"
.  The horrible truth slowly dawns, but
the others seem too drunk to pay any imm
ediate attention...": GOSUB 4630: GOSUB
4720: RETURN
```

```
4560 REM action nodes
4570 ON t(t,n,3) GOSUB 3030,3060,3070,30
80,3100: RETURN
4600 REM
4610 REM print messages if player presen
t
4620 REM
4630 IF FNc(c,2)=r THEN PRINT m$;
4640 m$="": RETURN
4650 REM
4660 REM select a message from data stat
ement
4670 REM
4680 RESTORE 7030: FOR m=1 TO t(t,n,4):
READ m$: NEXT m: RETURN
4690 REM
4700 REM print a blank line
4710 REM
4720 IF FNc(c,2)=r THEN PRINT
4730 RETURN
```

Listing Five

Get the Message

Line 4680 uses these DATA lines to retrieve messages. Unfortunately the Commodore computers cannot RESTORE to a line number, so a different technique must be used —see the next instalment for the appropriate flavours

```
7000 REM
7010 REM message data
7020 REM
7030 DATA "A strange smell fills the air
...could it be the odour of Catty-Kit A
La Carte?"," suddenly collapses on the f
loor, clutching "," looks rather ill, an
d warns the others not to touch the past
y."
7040 DATA " examines the tin carefully,
and assumes a thoughtfull expression."
```

Basic Flavours

The listing as printed will run on the Amstrad range of computers without modification. Flavours for other machines will be included in the next instalment

Listing Six

Directing The Cast

Each character is processed by the character handler in turn. These lines check the 'move' and 'handle' flags and call the appropriate routines to move characters or sort trees. Some REM statements have been indented to aid readability

```
1000 REM
1010 REM character handler
1020 REM
1030 REM check to see if key pressed
1040 GOSUB 4260: IF i$()"" THEN GOSUB 20
40: RETURN
1050     REM process each character in t
urn
1060 FOR c=1 TO 6
1070     REM check 'handle flag'
1080 IF FNc(c,10)>0 THEN c$(c,10)=FNm$(c
$(c,10),1): GOTO 1500
1090     REM flag=0 so reset flag and pr
ocess character
1100 RESTORE: FOR n=1 TO c%10+c-1: READ
c$(c,10): NEXT n
1110 IF FNc(c,10)=0 THEN GOTO 1500: REM
default value=0 so don't process
1120     REM check move flag
1130 IF FNc(c,11)>0 THEN c$(c,11)=FNm$(c
$(c,11),1): GOTO 1190
1140     REM move flag=0 so reset flag a
nd move character
1150 RESTORE: FOR n=1 TO c%11: READ c$(c
,11): NEXT n
1160 IF c$(c,11)="0" THEN GOTO 1180
1170 GOSUB 2840: GOTO 1500
1180     REM sort through trees
1190 GOSUB 2400: REM initialise conditio
ns
1200 t=2: GOSUB 5460: REM plot tree
1210 IF FNc(c,4)>0 THEN GOSUB 5000: REM
object manipulation tree
1500 NEXT c: GOTO 1030: REM do next char
 acter - when all finished check for key
 press/do again
```

PROBABLY A GOOD BET

Probability theory is at the heart of the scientific approach to gambling, yet it is virtually ignored by the majority of punters. By understanding the bookies' point of view, and relating all numbers to a common basis, a theorem such as Bayes' Rule can be easily implemented on your microcomputer.

Gambling is all about probability, yet most gamblers have only the haziest understanding of probability theory. This is why they will go on losing money (probably). Probability is a subject full of paradoxes and pitfalls, so it's not surprising that we find it difficult to grasp a theory which aims to be precise about uncertainty. But today's punter can turn to the computer for help. Computers are very good at calculating the odds. The modern gambler can go a long way with the aid of a microcomputer, a little self-discipline, and a background in elementary probability theory. The aim of this instalment is to provide that background, together with an example listing to illustrate some of the points discussed.

The first thing the well-informed punter should know is how to convert odds into probabilities and back again, yet surprisingly few do. There are two kinds of odds: odds in favour (F) and odds against (A). Bookmakers normally quote odds against an event or outcome, such as a horse winning a race, except when the event is said to be 'odds-on'. The picture is complicated by the practice of quoting a pair of integers such as 6 to 4 to define the odds, when 3 to 2 would seem more straightforward.

It's a good idea to simplify matters as far as possible, by reducing the varieties of odds to a uniform measure. This will help when we come to computerised calculations. The first step in converting odds to probabilities is to express the odds as a single number. This can be done using either formula below, where lowercase a (against) and f (for) are the two numbers quoted by the bookmaker.

$F = f/a$

$A = a/f$

Thus if the odds are given as 100 to 30 against, a = 100 and f = 30. So the odds in favour (F) are 30 / 100, which equals 0.30, and the odds against are 100/30 = 3.3333. Now we have reduced the odds to a common scale: thus 7 to 2 is 3.5 to 1, 100 to 30 is 3.333 to 1, 11 to 4 is 2.75 to 1, and so on. Already this makes comparisons easier.

The next step is to transform the odds into probabilities, with one of these formulae:

$P = F / (F+1)$

$P = 1 / (A+1)$

These give the probability of an event whose odds are F to 1 in favour or A to 1 against. So 100/30 or

Probability Theory

The serious study of mathematical probability began, as one might expect, with a series of losing bets. A certain Chevalier de Mere had succeeded, in the 17th century, in lining his pocket by repeatedly betting that he could, in four rolls of a die, throw at least one six.

Unfortunately, the Chevalier's success with this particular wager led to a distinct shortage of takers. He therefore switched, in 1654, to betting that in 24 tosses of a pair of dice he would throw a double-six at least once.

At this time, the only method of 'determining' probability in such cases was by throwing the dice as many times as possible and recording the result. This is not only tedious, but inaccurate as well, so when the Chevalier's new bet began to threaten him with bankruptcy he wrote to the mathematician Blaise Pascal for assistance. In this way the serious study of probability began.

Incidentally, the odds of throwing one six in four throws are 14 to 13 in favour. Using the formulae given in this article, can you see why the Chevalier's second wager lost him money?

DIGITISED IMAGE FROM 'HOW TO TAKE A CHANCE' BY DARRELL HUFF

3.3333 to 1 comes out as a probability of 1/4.3333 = 0.2308. Probability (P) is always expressed in terms of the event actually happening (such as a horse winning). If you want the probability of the event *not* occurring (Q), you can use the formula:

Q=1—P

Remember that P+Q=1 — the probability of an event happening plus the probability of that event not happening must total 1. Note also that long odds (against) are associated with unlikely events and therefore with small probabilities (in favour).

Bookmakers' odds are shorter than true odds; in other words the events concerned are made to seem more probable than they really are. The reasons for this become obvious when we look at the matter from the bookmaker's viewpoint. The bookmaker has to make a living and cover his expenses. Accordingly, he creates what is called in the jargon an 'over-round' book. An imaginary horse-race illustrates this:

Runner	Bets Taken	Odds	Liability
Apricot Jam	£1,000	5-6	£1,833.33
Atari Sayonara	£750	13-8	£1,968.73
Small Acorn	£200	8-1	£1,800.00
Red Apple	£50	33-1	£1,700.00
	£2,000		

In this case, £2,000 has been wagered, half of it on Apricot Jam, and the rest as indicated. The odds have been chosen to keep the bookmaker's liability below the total staked, namely £2,000, whatever happens. Thus if Small Acorn wins, the punters who backed it to the tune of £200 will get their stakes returned, plus eight times the stakes (£1,600) as winnings — making £1,800 in all. This leaves £200 profit (10 per cent) for the bookie. In a fair book, Apricot Jam would be even money and Small Acorn would start at 9 to 1. But who would want to run a fair book?

One thing about our imaginary book, which is true to life, is that the outsiders are given relatively poor odds. This means that the bookie generally recall that long odds (against) translate into low probabilities (in favour) you'll see that the bookmakers shorten the odds to make their percentage. Remember that the bookies' odds tend to be shorter than the true odds. To make money you have to find cases where they have not shortened them enough.

To save you some paperwork (or brainwork) our listing is a program that takes the drudgery out of these calculations. It also computes the bookies' margin. We show two sample runs, one on the game mentioned above, the second on the 1985 Derby (with starting prices) run at Epsom

Odds To Probability

Let's work through an example of odds to probability conversion. Below are the odds quoted on a fixed-odds coupon for a football game, Leicester versus Everton.

15/8 Leicester 5/2 Everton 1/1

What this says is that the odds against the home side (Leicester) winning are 15 to 8; the odds against a draw are 5 to 2, and the odds against an away win by Everton are 1 to 1, or evens. (Incidentally, this match took place in the summer of 1985 — Everton took the lead, but in the end Leicester won by three goals to one, so the underdogs don't always lose!)

First, we standardise the odds: A=a/f.

Home	15/8	=	1.875
Draw	5/2	=	2.50
Away	1/1	=	1.0

Next, we transform into probabilities: P=1/(A+1).

Home	1/2.875	=	0.3478
Draw	1/3.5	=	0.2857
Away	1/2	=	0.5000
Total		=	1.1335

Notice that these probabilities add up to more than one — 1.1335 to be precise. This breaks the laws of probability, but it is not illegal. It just means that the bookies' margin is 13.35% on this match. If you

```
 10 REM ******************************
 15 REM ** PROBABILITY CALCULATIONS **
 20 REM ******************************
100 PRINT'"Probability Calculator:"
110 PRINT "Please give odds as two numbers;"
120 PRINT "e.g. 3,1 for 3 to 1 etc."
130 PRINT "Use 0,0 to end input."'
140 at%=&0102040A : REM format
150 N=0
160 TP=0 : TA=0
190 REM -- Main Loop:
200 REPEAT
202   @%=4
210   N=N+1
220   PRINT "Runner ";N;" name is ";
222   INPUT NAME$
225   PRINT "Odds against are ";
230   INPUT A,F
240   IF A<=0 AND F<=0 THEN GOTO 310 : REM quit
250   P = F/(F+A)
260   A=A/F
270   TP=TP+P : REM total prob.
280   TA=TA+A : REM total odds.
290   PRINT "For runner no. ";N;", ";NAME$
295   @%=at%
296   PRINT "probability is :  ";P
300   PRINT
310   UNTIL A<=0 OR F<=0
320 N=N-1
322 @%=at%
330 PRINT
333 PRINT "Stake returned is ";100/TP;"%"
335 PRINT "Bookies' mark-up: ";100*(TP-1);"%"
360 IF TP<1 THEN PRINT "You're kidding!";CHR$7
999 END
>RUN
```

does better when a long shot wins — so beware of backing 'dark horses'. You are likely to be subsidising more popular choices. This is simply because punters are reluctant to go much below even money, so the bookies are constrained in the odds they can offer on hot favourites. They compensate by clawing back on the 'no-hopers'.

From time to time the bookies fail to cover their liabilities, either because they react too slowly to sudden swings in the betting, or because they open with unrealistic prices, though such events are rare. Don't get the impression that you can beat the bookies, because you can't. What you may be able to do is get some money from your fellow punters, with the bookie acting as middle-man and taking his cut.

In the next instalment, we'll focus on a theorem designed to calculate the probability of an event occurring — Bayes' Rule — which we'll apply to forecasting football results. We'll also be looking at some famous and infamous betting 'systems', as well as considering what recent developments in artificial intelligence have to offer the computer-based punter.

Diving Into Pools

Punters fill out pools coupons in many ways, but one of the most popular methods is the '8 from 10 full perm' where the punter puts crosses beside ten matches. When the results are known, the best eight selections are taken to maximise the number of pools point. Methods of selecting matches vary greatly but the chance of choosing eight score draws randomly is extremely remote.

```
Probability Calculator:
Please give odds as two numbers;
e.g. 3,1 for 3 to 1 etc.
Use 0,0 to end input.

Runner 1 name is ?home
Odds against are ?15,8
For runner no. 1, home
probability is :  0.3478

Runner 2 name is ?draw
Odds against are ?5,2
For runner no. 2, draw
probability is :  0.2857

Runner 3 name is ?away
Odds against are ?1,1
For runner no. 3, away
probability is :  0.5000

Runner 4 name is ?
Odds against are ?0,0

Stake returned is 88.2192%
Bookies' mark-up: 13.3540%
>
>
>RUN



Probability Calculator:
Please give odds as two numbers;
e.g. 3,1 for 3 to 1 etc.
Use 0,0 to end input.

Runner 1 name is ?Slip Anchor
Odds against are ?9,4
For runner no. 1, Slip Anchor
probability is :  0.3077

Runner 2 name is ?Law Society
Odds against are ?5,1
For runner no. 2, Law Society
probability is :  0.1667

Runner 3 name is ?Damister
Odds against are ?16,1
For runner no. 3, Damister
probability is :  0.0588

Runner 4 name is ?Supreme Leader
Odds against are ?10,1
For runner no. 4, Supreme Leader
probability is :  0.0909

Runner 5 name is ?Lanfranco
Odds against are ?14,1
For runner no. 5, Lanfranco
probability is :  0.0667

Runner 6 name is ?Reach
Odds against are ?33,1
For runner no. 6, Reach
probability is :  0.0294

Runner 7 name is ?Theatrical
Odds against are ?10,1
For runner no. 7, Theatrical
probability is :  0.0909

Runner 8 name is ?Phardante
Odds against are ?40,1
For runner no. 8, Phardante
probability is :  0.0244

Runner 9 name is ?Royal Harmony
Odds against are ?40,1
For runner no. 9, Royal Harmony
probability is :  0.0244

Runner 10 name is ?Snow Plant
Odds against are ?100,1
For runner no. 10, Snow Plant
probability is :  0.0099

Runner 11 name is ?Petowski
Odds against are ?33,1
For runner no. 11, Petowski
probability is :  0.0294

Runner 12 name is ?Seurat
Odds against are ?33,1
For runner no. 12, Seurat
probability is :  0.0294

Runner 13 name is ?Shadeed
Odds against are ?7,2
For runner no. 13, Shadeed
probability is :  0.2222

Runner 14 name is ?Main Reason
Odds against are ?200,1
For runner no. 14, Main Reason
probability is :  0.0050

Runner 15 name is ?
Odds against are ?0,0

Stake returned is 86.5215%
Bookies' mark-up: 15.5781%
>
```

CONFINED TO CELLS

Having written the graphics-handling and formula-processing routines for our spreadsheet program, we are now in a position to add the routines that allow formulae and data to be entered in the sheet and the calculations to be made.

The code that handles formula input can be found between lines 2000 and 2050. This section of code used a BASIC INPUT command to get a formula from the user. This formula is, of course, in infix form and is initially stored in D$. The routine then goes on to store the formula in the array F$(). At this stage the corresponding entry in the array that holds the reverse Polish version, PS$(), is cleared.

The formulae for cells A1 to A15 are held in F$(1) to F$(15); cells B1 to B15 are held in F$(16) to F$(30) and so on. The correct element in F$() can be found at any stage from the spreadsheet cursor co-ordinates using the formula (Y-1)*15+X.

The next routine, between lines 2100 and 2250, deals with entering data to a cell on the sheet. This subroutine is called from the main keyboard scanning routine at line 1170 if a numeric key is pressed. The routine prints a message to indicate that you're entering data into the current cell and then takes data a character at a time at line 2120. If the key pressed is one of the cursor keys or the Space bar, then the routine is terminated and the numeric data that has been gathered so far is placed in an array M(,). Lines 2149 to 2170 check to ensure that the data entered is valid and, if it is, adds the number to E$. One limitation of our spreadsheet program is that each cell can accept a maximum of only five characters; hence extra input is checked for at line 2160 and a message is printed at line 2220.

CALCULATING THE SHEET

The calculate routine starts at line 2300 and acts as a control loop for the reverse Polish translation routines developed in the last two instalments before actually evaluating the formulae in the subroutine at line 2370.

The first routine works its way through the infix formulae array, F$(), and the corresponding reverse Polish array PS$(), translating any newly entered formulae into reverse Polish by calling the subroutine at line 4000. If there is an entry in the current element being tested then the evaluation routine is called.

The evaluation routine makes use of the reverse Polish legality check at line 4700, which deposits the elements of the reverse Polish string it is working on into an array, G$(). These elements are either operators (such as + and –) or operands (A1, B5, 3 and so on). The evalution routine then works on the elements of the reverse Polish string according to the following conditions:

- If the element is a constant (differentiated from a cell address by the fact that the left-hand character will be numeric rather than alphabetic) then the value is put into the array C() and its position in C() is put onto a stack, ST().
- If the element is a cell address, then its value is found from the array M(,) and put into C(). Its position in C() is placed on the stack.
- If the element is an operator then the operation is carried out on two (or in the case of unary minus, one) previously stacked operands. The stack in fact holds the positions of the operands in C() and so the numbers that are taken from the stack are used to locate the correct elements in C(). The result is put into C() and the two operands are removed from the stack. Then the position of the result in C() is put onto the stack.

After the complete string has been processed, the result of evaluating the string will be held in the last element of C() and can be transferred to the appropriate element of M(,), the array that holds all the cell values.

Basic Flavours

Amstrad CPC 464/664:
Make the following changes to the Commodore 64 version:

```
2010 LOCATE 1,22
2020 PRINT "NEW FORMULA          ";CHR$(11)
2103 LOCATE 1,22
2120 A$="":WHILE A$="":A$=INKEY$:WEND
2130 IF A$=CHR$(243)OR A$=CHR$(242) OR
     A$=CHR$(241) OR A$=CHR$(240) THEN
     2250
2180 LOCATE H(X+1-H1) -1,V(Y-V1+1)
2190 PRINT CHR$(24) ;SPACE$((5) ;CHR$(11)
2200 LOCATE H(X+1-H1) +4-LEN(E$), V(Y-V1+1):
     PRINT E$;CHR$(24)
2220 LOCATE 1,22
2305 LOCATE 1,22
2315 Q$=CHR$ (J+64) +MID$ (STR$(I), 2,2):
     LOCATE 1,1:PRINT "CELL:";Q$;" "
```

BBC Micro:
Make the following changes to the Commodore 64 version:

```
2010 PRINT TAB (0,22);
2103 PRINT TAB (0,22);
2120 A$=get$
2130 IF A$=CHR$(136) OR A$=CHR$(137) OR
     A$=CHR(138) OR A$=CHR$(139) THEN 2250
2180 COLOUR 2:COLOUR 129
2190 PRINT TAB(H(X+1-H1) -1,V(Y-V1+1)-1);"  "
2200 PRINT TAB(H(X+1-H1) +4-LEN(E$),
     V(Y-V1+1)-1);E$
2205 COLOUR 1:COLOUR 128
2220 PRINT TAB(0,22);
2230 I$=GET$
2240 PRINT TAB(0,22);
2305 PRINT TAB(0,22);
2315 Q$=CHR$ (J+64) +MID$(STR$(I) ,2,2):PRINT
     TAB (0,0);"CELL:" ;Q$;"  "
```

Entry And Calculation Routines

Commodore 64:

```
1170 IF A$>="0" AND A$=<"9" THEN GOSUB 2
100:REM ENTER NUMERIC DATA
2000 REM *** INPUT FORMULA ROUTINE ***
2010 GOSUB 1950:REM MOVE CURSOR TO INPUT
 LINE
2020 PRINT "NEW FORMULA:
          ";CU$
2030 INPUT "NEW FORMULA:";D$
2040 LET F$((Y-1)*15+X)=D$:LET PS$((Y-1)
*15+X)=""
2050 GOSUB 1900:RETURN
2100 REM *****  ENTER DATA IN CELL *****
2103 GOSUB 1950:REM MOVE CURSOR TO INPUT
 LINE
2104 PRINT "            ENTERING -DATA
          "
2105 LET P=0:LET E$=""
2110 IF A$<>"" THEN 2150
2120 GET A$:IF A$="" THEN 2120
2130 IF A$=CHR$(29) OR A$=CHR$(157) OR A
$=CHR$(17) OR A$=CHR$(145) THEN 2250
2135 IF A$=" " THEN 2250
2140 IF A$="." THEN 2160
2150 IF A$<"0" OR A$>"9" THEN 2120
2160 LET P=P+1:IF P>5 THEN 2220
2170 LET E$=E$+A$
2180 PRINT CO$;:FOR C=1 TO V(Y+1-H1)-1:P
RINT CD$;:NEXT C
2190 PRINT TAB(H(X+1-H1)-1);CHR$(18);"
   ";CU$
2200 PRINT TAB(H(X+1-H1)+4-LEN(E$));CHR$
(18);E$
2210 GOTO 2120
2220 GOSUB 1950:REM MOVE CURSOR TO INPUT
 LINE
2225 PRINT "      ERROR - INPUT IGNORED"
2230 GET I$:IF I$="" THEN 2230
2240 PRINT CO$;:FOR K=1 TO 22:PRINT CD$;
:NEXT K
2245 PRINT "
";:GOTO 2120
2250 LET M(Y,X)=VAL(E$):GOSUB 1900:RETUR
N
2300 REM ****** CALCULATE THE SHEET ****
***
2305 GOSUB 1950:REM MOVE CURSOR TO INPUT
LINE
2306 PRINT "      CALCULATING  -  PLEASE
 WAIT     "
2310 FOR J=1 TO 15:FOR I=1 TO 15
2315 Q$=CHR$(J+64)+MID$(STR$(I),2,2):PRI
NT CO$;CD$;"CELL:";Q$;" "
2320 LET P$=PS$((J-1)*15+I)
2325 IF P$<>"" THEN GOSUB 4700:GOSUB 237
0:GOTO 2350
2330 LET C$=F$((J-1)*15+I)
2335 IF C$="" THEN 2350
2340 GOSUB 4000:GOSUB 2370:REM      DECO
DE FORMULA
2350 NEXT I,J:GOSUB 1700:RETURN
2370 REM *****  EVALUATE FORMULA  ******
2375 SP=0:FOR K=1 TO 20:ST(SP)=0:NEXT K
2380 FOR K=1 TO CP
2390 LET T$=G$(K)
2400 IF T$="+" THEN 2500
2405 IF T$="-" THEN 2510
2410 IF T$="*" THEN 2520
2415 IF T$="/" THEN 2530
2420 IF T$="^" THEN 2540
2425 IF T$="&" THEN 2550
2430 IF T$="" THEN 2450
2432 TE$=MID$(T$,1,1)
2435 IF (TE$>="0" AND TE$<="9") OR TE$="
." THEN C(K)=VAL(T$):GOTO 2445
2440 LET C(K)=M(ASC(MID$(T$,1,1))-64,VAL
(MID$(T$,2,2)))
2445 SP=SP+1:ST(SP)=K
2450 NEXT K
2460 LET M(J,I)=C(K-1):RETURN
2500 C(K)=C(ST(SP-1))+C(ST(SP)):ST(SP)=0
:SP=SP-1:ST(SP)=K:GOTO 2450
2510 C(K)=C(ST(SP-1))-C(ST(SP)):ST(SP)=0
:SP=SP-1:ST(SP)=K:GOTO 2450
2520 C(K)=C(ST(SP-1))*C(ST(SP)):ST(SP)=0
:SP=SP-1:ST(SP)=K:GOTO 2450
2530 C(K)=C(ST(SP-1))/C(ST(SP)):ST(SP)=0
:SP=SP-1:ST(SP)=K:GOTO 2450
2540 C(K)=C(ST(SP-1))^C(ST(SP)):ST(SP)=0
:SP=SP-1:ST(SP)=K:GOTO 2450
2550 C(K)=0-C(ST(SP)):ST(SP)=K:GOTO 2450
3010 DIM H(5),V(8),ST(20),ST$(20),E$(20)
,G$(20),C(20)
```

Sinclair Spectrum:

```
2000>REM ************************
2001 REM * INPUT FORMULAE       *
2002 REM ************************
2010 PRINT AT 18,0;"       ENTER
NEW FORMULA          "
2020 INPUT  LINE D$
2030 LET F$((Y-1)*15+X,1 TO LEN
D$)=D$: LET H$((Y-1)*15+X,1 TO )
=""
2050 GO SUB 1900
2060 RETURN
2100 REM ************************
2101 REM * ENTER DATA IN CELL   *
2102 REM ************************
2105 PRINT AT 18,0;"       ENTERI
NG - DATA            "
2110 LET P=0: LET B$=""
2120 LET A$=INKEY$: IF A$="" THE
N  GO TO 2120
2130  IF A$=CHR$ (13) THEN  GO S
UB 1650: GO TO 2250
2140 IF A$="." THEN  GO TO 2160
2150 IF A$<"0" OR A$>"9" THEN  G
O TO 2120
2160 LET P=P+1: IF P>5 THEN  GO
TO 2220
2170 LET B$=B$+A$
2180 PRINT AT V(Y+1-V1),H(X+1-H1
);"      "
2190 PRINT AT V(Y+1-V1),H(X+1-H1
)+5-LEN B$;B$
2210 GO TO 2120
2220 PRINT AT 18,0;"     ERROR -
INPUT IGNORED        "
2230 LET I$=INKEY$: IF I$="" THE
N  GO TO 2230
2240 PRINT AT 18,0;"
                     ": GO TO 211
2250 LET M(Y,X)=VAL (B$): GO SUB
 1900
2260 RETURN
2300 REM ************************
2301 REM * CALCULATE THE SHEET *
2302 REM ************************
2305 PRINT AT 18,0;"      CALCULA
TING - PLEASE WAIT   "
2310 FOR J=1 TO 15: FOR I=1 TO 15
2315 PRINT AT 0,0;"CELL:";CHR$ (
J+64);STR$ (I);" "
2320 LET P$=H$((J-1)*15+I,1 TO )
2325 IF P$(1)<>" " THEN  GO SUB
2355: GO SUB 4700: GO SUB 2370:
GO TO 2350
2330 LET C$=F$((J-1)*15+I,1 TO )
2335 IF C$(1)=" " THEN  GO TO 2350
2340 GO SUB 4000: GO SUB 2370
2350 NEXT I: NEXT J: GO SUB 1700
: RETURN
2355 FOR Z=1 TO LEN P$: IF P$(Z)
=" " THEN  GO TO 2357
2356 NEXT Z
2357 LET P$=P$(1 TO Z-1): RETURN
2370 REM ************************
2371 REM * DECODE FORMULA       *
2372 REM ************************
2375 LET SP=0: DIM S(20)
2380 FOR K=1 TO CP
2390 LET T$=G$(K)
2400 IF T$(1)="+" THEN  GO TO 2500
2405 IF T$(1)="-" THEN  GO TO 2510
2410 IF T$(1)="*" THEN  GO TO 2520
2415 IF T$(1)="/" THEN  GO TO 2530
2420 IF T$(1)="^" THEN  GO TO 2540
2425 IF T$(1)="&" THEN  GO TO 2550
2430 IF T$(1)="" THEN  GO TO 2450
2432 LET U$=T$(1)
2435 IF (U$)="0" AND U$(="9") OR
 U$="." THEN  LET C(K)=VAL (T$):
 GO TO 2445
2440 LET C(K)=M(CODE (T$( TO 1))
-64,VAL (T$(2 TO )))
2445 LET SP=SP+1: LET S(SP)=K
2450 NEXT K
2460 LET M(J,I)=C(K-1): RETURN
2500 LET C(K)=C(S(SP-1))+C(S(SP)
): LET S(SP)=0: LET SP=SP-1: LET
 S(SP)=K: GO TO 2450
2510 LET C(K)=C(S(SP-1))-C(S(SP)
): LET S(SP)=0: LET SP=SP-1: LET
 S(SP)=K: GO TO 2450
2520 LET C(K)=C(S(SP-1))*C(S(SP)
): LET S(SP)=0: LET SP=SP-1: LET
 S(SP)=K: GO TO 2450
2530 LET C(K)=C(S(SP-1))/C(S(SP)
): LET S(SP)=0: LET SP=SP-1: LET
 S(SP)=K: GO TO 2450
2540 LET C(K)=C(S(SP-1))^C(S(SP)
): LET S(SP)=0: LET SP=SP-1: LET
 S(SP)=K: GO TO 2450
2550 LET C(K)=0-C(S(SP)): LET S(
SP)=K: GO TO 2450
3010>DIM H(4):DIM V(7):DIM S(20)
:DIM S$(20,5):DIM E$(20,5):DIM G
$(20,5):DIM C(20)
```

NEW TRICKS FOR AN OLD LANGUAGE

Developed in the 1950s, FORTRAN lacks many of the structures and facilities we have become accustomed to in more modern programming languages. We look at a few techniques adopted by programmers to overcome FORTRAN's limitations.

There have been two major criticisms of FORTRAN as a modern programming language: its lack of proper control structures, which makes it difficult to implement programs in a way that can be easily understood and amended, and the very restricted facilities for character handling. Over the years, a number of ways of overcoming these deficiencies have been tried.

A relatively recent approach is to use a 'preprocessor' — a sort of higher level compiler. Programs written in an extended version of FORTRAN are then run through the preprocessor, which changes all the extra features into equivalents in standard FORTRAN. This gives the advantage of being able to write programs in a properly structured manner while maintaining a standard and hence portable version. The best known examples of FORTRAN preprocessors are WATFOR and RATFOR.

When the FORTRAN 77 standard was issued, it incorporated new features in a similar way to the popular preprocessors. For this reason, there remains a fair degree of consistency in the language, although many compilers for micros are based on FORTRAN IV with extensions, rather than on true FORTRAN 77.

The problem with providing proper control structures is that FORTRAN does not have a block structure like ALGOL, PASCAL or C, and there is no way of producing a compound statement by bracketing a number of statements together (with begin. . . .end, for example). The only way a block can be isolated is to extract it as a separate subroutine or to surround it with GOTOs. One new control structure has been introduced, though it doesn't fit in very well with the rest of the language. This is an IF. . .THEN. . .ELSE. . .ENDIF:

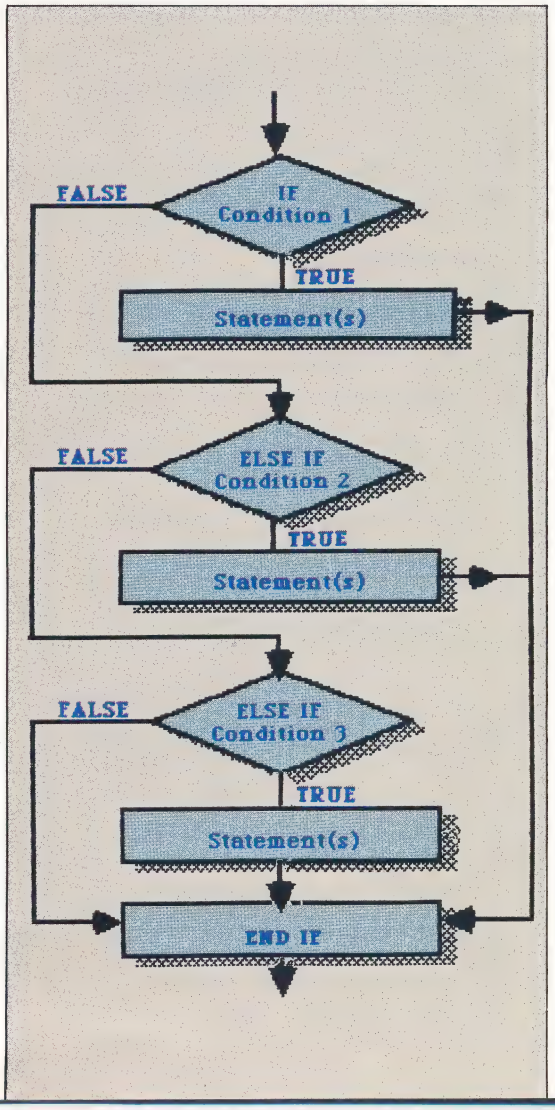
```
IF (logical expression) THEN
        .........
        statements
        .........
ELSE
        .........
        statements
        .........
ENDIF
```

The logical expressions are formed in exactly the same way as in standard FORTRAN and as is usual the ELSE part may be omitted.

A very useful form of this statement, which other languages would do well to copy, is a variation to cope with nested IFs:

```
IF (logical expression) THEN
        .........
        statements
        .........
ELSEIF
        .........
        statements
        .........
ELSEIF
        .........
ELSE
        .........
        statements
        .........
ENDIF
```

where there may be as many ELSEIFs as required.



Nest of IFs
Most versions of BASIC allow nested IF statements, but FORTRAN goes one better by implementing the ELSEIF construct, which enables IF statements to be nested to any depth over a number of program lines. The construct is terminated by an ENDIF statement

The second major enhancement has been the introduction of a character data type. Character strings are declared at the start of the program with the other declarations in one of the two forms:

```
CHARACTER*9                    CH1,CH2
CHARACTER                      CH3*7,CH4*16
```

Note that a maximum length for a string must be given as *n, where n is an integer. This may be applied either to the CHARACTER key word itself if all the strings are the same length, or individually to each named string. These declarations give two strings of length 9, one of length 7, and one of length 16.

Arrays of strings may be declared in the normal way:

```
CHARACTER*4 CHARS(50)
```

which declares an array of 50 four-character strings. String constants can be assigned to string variables as in:

```
CH1'ABCDEFGHI'
```

If the assigned string is too short, blanks will be appended; if it is too long, the excess will be truncated.

Strings can be compared using the normal relational operators, such as .GT.,.LT. and so on, but there are also two new operators to carry out string functions. The substring operator gives access to any sequence of characters from the string, as in:

```
CH1(3:5)
```

MIKE CLOWES

Cracking The Code

Elementary My Dear
This FORTRAN program reads in data from a file called SECRET and uses a decoding key to decipher encoded text

```
C     PROGRAM TO DECODE SECRET
C     MESSAGES NOTE PRESENCE OF A
C     'PROGRAM' STATEMENT IN FORTRAN 77
C
      INTEGER COUNT,PTRIN,PTROUT
      LOGICAL FILE
      CHARACTER*10 NUM,CHAR
      CHARACTER*30 IN,OUT
      DATA COUNT /0/
C
C     NOTE 'DATA' STATEMENT TO INITIALISE
C     A VARIABLE
C
C     IF SECRET FILE DOES NOT EXIST PRINT
C     ERROR MESSAGE
C
      INQUIRE(FILE='SECRET',EXIST=FILE)
      IF(.NOT.FILE)THEN
        WRITE(1,10)
        STOP
      ENDIF
C
C     OPEN FILES
C
      OPEN(UNIT=12,FILE='MESSAGE',STATUS
        ='NEW')
C
C     SPECIFY DECODER KEY
C
      NUM='0123456789'
      CHAR='MW3 OD$%^ T'
C
C     READ AND DECODE SECRET MESSAGE
C
100   READ(11,20,END=1000)IN
        OUT=' '
        PTROUT=1
        DO 200 I=1,30
          PTRIN=INDEX(CHAR,IN(I:I))
C
C     STANDARD FUNCTION 'INDEX'
C     DETERMINES THE POSITION
C     OF SUBSTRING IN(I:I) IN STRING CHAR
C
          IF(PTRIN.NE.0)THEN
            OUT(PTROUT:PTROUT)=NUM
            (PTRIN:PTRIN)
            PTROUT=PTROUT+1
          ENDIF
200   CONTINUE
      WRITE(12,20)OUT
C
C     NOTE USE OF SAME FORMAT FOR INPUT
C     AND OUTPUT
C
        COUNT=COUNT+1
      GOTO 100
C
C     CLOSE DOWN
C     'ENDFILE' IS USED TO WRITE END OF
C     FILE MARKER
C
1000  ENDFILE(UNIT=12)
      STOP
C
C     FORMATS
10    FORMAT(1H ,'SECRET FILE DOES NOT
      EXIST')
20    FORMAT(A)
C
C     NOTE THAT THE NUMBER OF
C     CHARACTERS TO BE INPUT NEED NOT
C     BE SPECIFIED
C
      END
```

This gives the substring of CH1 starting at the third character and ending at the fifth, so:

```
CH1='ABCDEFGHI'
CH2=CH1(3:5)
```

would result in CH2 containing 'CDE', or strictly 'CDE '. Note that the second figure given is the position of the final character and not the length, which would be more familiar to a BASIC programmer.

The concatenation operator, //, is used to join strings together, as in:

```
CHARACTER CH1*4,CH2*4, CH3*8
CH1='ABCD'
CH2='WXYZ'
CH3=CH1//CH2
```

which would leave CH3 containing 'ABCDWXYZ'.

One subject we have yet to look at is the use of data files. Taking full advantage of FORTRAN requires a system with disk drives and most FORTRANS will include facilities for both sequential and direct access to files. The actual source or destination of an I/O operation is determined by the device identifier in the READ or WRITE statement. Some numbers, usually the lower ones, are reserved for standard I/O devices such as the keyboard, screen and printer; however, there will be a range of numbers available to the programmer. The OPEN statement is used to assign an identifier to a disk file. It takes the form:

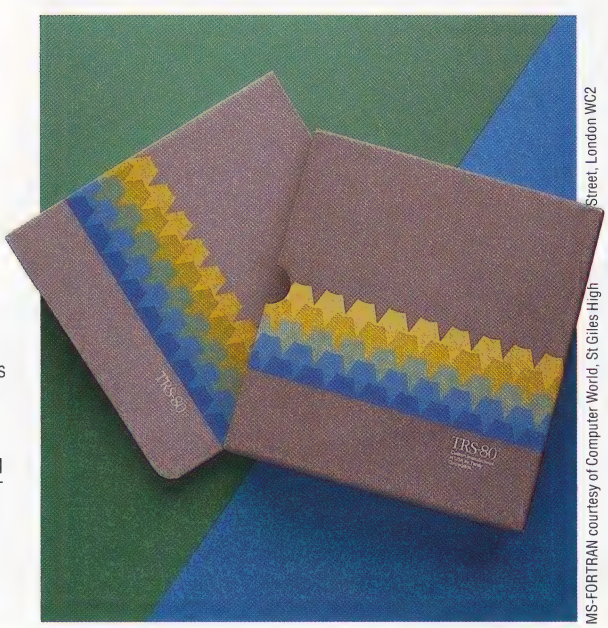
```
OPEN (UNIT=integer expression,FILE=file name,
STATUS=status)
```

The integer value for the UNIT is the number that should be used in subsequent READ or WRITE statements to access the file, and the file name would be according to the operating system conventions. The status can be one of several values — for example, OLD for an input file that is already in existence or NEW for an output file that is being created.

FORTRAN For Micros

FORTAN compilers tend to be rather expensive and are normally available only for business machines — the package illustrated cost £130 and runs on the Tandy 2000. Furthermore, limited memory space often makes micro implementation of FORTRAN difficult or even impossible.

However, with the increasing availability of CP/M on home computers such as the Commodore 128, Amstrad CPC machines and others, many home users will find that there is a version of FORTRAN available for their machine. Amstrad users in particular are well catered for — Nevada FORTRAN costs only £39.95 including VAT from New Star Software Ltd, 45 Plovers Mead, Wyatts Green, Essex, CM15 0PS



MS-FORTRAN courtesy of Computer World, St Giles High Street, London WC2

There are a number of optional clauses in the OPEN statement if something other than a straightforward sequential file is required. These are: ACCESS=, to determine sequential or direct access; FORM=, to determine whether the file is formatted or unformatted; IOSTAT= which provides a means of error recovery if the proper file assignment cannot be made for some reason; and RECL=, to specify a record length.

A similar CLOSE statement might appear as:

```
CLOSE(UNIT=integer expression,STATUS=status)
```

This can be used to close an opened file but isn't always necessary, since files will be automatically closed when the program terminates.

In addition, there are a number of other file handling statements that we haven't mentioned. Perhaps the most useful of these is INQUIRE, which enables the programmer to determine the current status and activity of any file.

There are also a number of useful additions to the READ and WRITE statements, for use when doing I/O operations with files. For example:

```
READ(7,10,REC=I)A,B,C
```

would read the Ith record in a file that had been opened for direct access. As a further example:

```
READ(7,11,END=1000)A,B,C
```

would cause control to transfer to statement number 1000 when the end of file is encountered on a sequential file.

Finally, we'll look at the FORTRAN compilers themselves. The first thing to note is that they are usually fast and efficient in operation and produce fast, compact and efficient code, which is a major advantage particularly for real-time applications. This is perhaps as it should be for a language that has been around for so long and is fairly low level to begin with.

Error detection, however, is not a strong point of FORTRAN compilers. They can quite happily accept some horrendous errors that would never get past, say, a PASCAL compiler. Spaces, for example, are not significant in a FORTRAN statement, so a compiler will often remove all spaces from each line as a first task. One example is the DO statement:

```
DO 100 I=1,100
```

This sets up a loop to be repreated 100 times. A full stop in place of the comma, however, after removing the spaces, leaves you with:

```
DO100I=1.100
```

which is a perfectly legitimate assignment to a variable called 'DO100I! This simple error is rumoured to have led to an extremely expensive disaster in the US space program. FORTRAN has been the standard language for much defence work, and errors such as this one, occurring in (among other things) nuclear early warning systems, were major factors in the development of ADA as a replacement language.

DESIRABLE RESIDENTS

In this instalment of our series on the Amstrad operating system, we look at the provision of resident system extensions, which allow new commands to be added to Locomotive BASIC, and discuss foreground, background, and extension ROMs.

At the heart of the Amstrad operating system is the kernel. It is this section of the firmware that takes care of all the internal housekeeping of the computer, such as processing events and interrupts, and memory management.

Most entries to the kernel are made via addresses between &BCC8 and &BD10, but the kernel is also provided with its own jumpblock, divided into an upper and lower section, lying at &B900-&B921 and &0000-&003B respectively. Unlike the main firmware jumpblock area, however, these locations should not be patched by the user. Some of the entries are of considerable use, and a brief description of the more interesting ones may be found in the Useful Kernel Addresses Table.

One very powerful feature of the kernel is its ability to process external commands. These commands are held either in ROMs or in RAM. In the case where they are loaded into RAM, they are known as resident system extension (RSXs). An external command can be a routine of any size or function — for example, the commands provided with AMSDOS such as |DIR and |ERA are routines that handle the disk files. Additionally, the whole of the BASIC language appears to the system as an RSX — try typing |BASIC and see what happens.

INTRODUCING COMMANDS

External commands may be set up in two ways. First, on power-up, where the commands are loaded in from ROM; second, from under program control, where the commands may be loaded from either ROM or RAM. In both cases, an external command is referenced by a command table, the format of which is detailed in the diagram. Command names can be up to 16 characters long, with the last character having bit 7 set. The characters can be any seven-bit code, but if the command is to be called from BASIC care should be taken to ensure that the characters used are accessible from the keyboard.

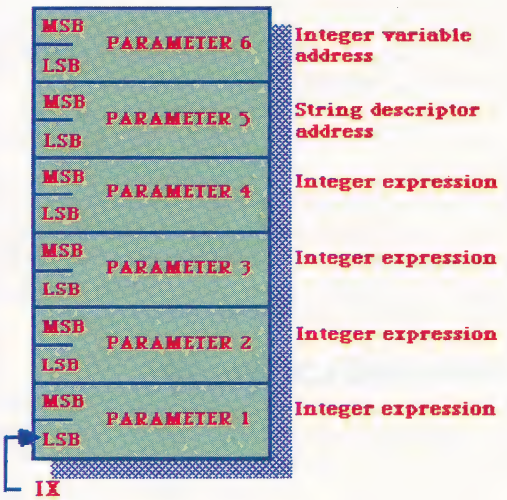
If the commands are in the form of an RSX (that is to say loaded into RAM) then the actual process of 'logging on' — alerting the firmware to their presence — is done using the kernel routine KL_LOG_EXT. This routine is called via &BCD1, with the address of the command table in BC. HL must contain the address of a four-byte workspace area that can be used by the kernel, and both addresses must lie in the central 32 Kbytes of RAM.

In cases where commands are stored in ROM, it is the ROM that is introduced to the firmware rather than the command table. The ROM must also conform to a particular format, as shown in the third diagram. ROMs may be designated as foreground, background, or extension ROMs. Foreground ROMs are used to contain programs that, when entered, take over control of the firmware. For example, the BASIC ROM is of this type, as would be other language ROMs, such as PASCAL.

A background ROM contains external commands that may or may not be logged on by the foreground program. Finally, an extension ROM is used to contain code for either a foreground program or an external command routine that would not fit into the relevant ROM.

Each ROM is addressed in the range 0-251, although different types are restricted in different ways. Foreground ROMs may occupy any address, provided that all addresses below the one chosen are occupied by other ROMs of any type, since the kernel searches for ROMs by stepping up from address 0 until it finds the first empty space. Background ROMs should be fitted at addresses 1-7 on the 464, and 1-15 on the 664 and 6128. Extension ROMs may occupy any address.

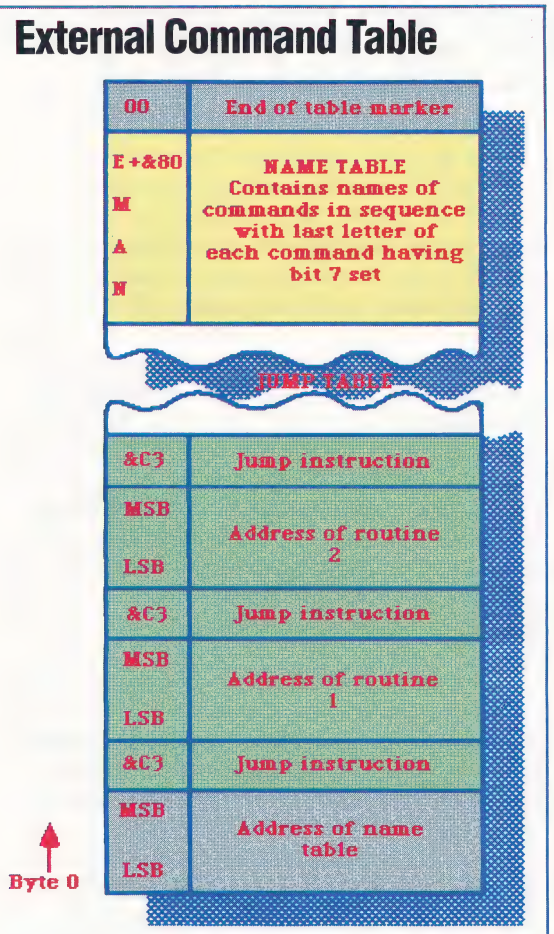
Parameter Table Addressing



Entry Process
On entry to a command routine called from BASIC, any parameters passed by the user are stored in a parameter block. The base address of the block is pointed to by IX and a A register holds the number of parameters passed. The diagram shows the structure of the block set up after executing |COMMAND, 5,24.6,a%,x,@b$,@ x%

CAROLINE CLAYTON

Setting The Table
External commands are referenced via a command table. A table may contain data for a number of different commands grouped together and the format of such a table is shown here



For example, if the system had FORTH as ROM 0 (default) and a background ROM at ROM 1, then a second foreground ROM should be installed at ROM address 2 to ensure continuity. A second background ROM, however, could be introduced at any address between 3 and 7 (15 on the 664/6128), and an extension could be installed at any address.

All foreground ROMs are logged on when the machine is first powered up. Background ROMs may then be either initialised individually, using KL_INIT_BACK, or all at once, using KL_ROM_WALK. The BASIC ROM, incidentally, initialises all background ROMs when it is entered.

Any extension ROM occupies the same area of memory as the on-board BASIC ROM (&C000 to &FFFF). Before any ROM can be entered, the BASIC ROM must first be disabled and the appropriate ROM itself enabled. The kernel provides several routines to switch both the upper and lower ROMs in and out, which relieves the programmer of the need to switch the relevant hardware directly.

COMMAND LOCATION

When the external commands have been logged on, they can be accessed directly from BASIC (see below) or via the kernel using the KL_FIND_COMMAND entry. This entry takes a command name and searches the RSXs and background ROMs for a match. If a command is found then the routine's entry address is returned in HL, with the C register containing the background ROM number (this can be ignored if the command routine lies in RAM). If no command is found, the routine returns with carry false. This routine is called at &BCD4, with HL containing the address of the command name, which should be terminated by a character with bit 7 set.

External commands can be entered simply from BASIC by preceding the command name with a | symbol (shifted @). BASIC also allows parameters to be passed to the routine by separating them with commas. For example:

```
|COMMAND,1,2,3,x,y*3,@$,@X%
```

The firmware then passes control to the command routine, with the number of parameters held in the A register and IX pointing to a table containing the parameter information. This table contains two bytes for each parameter, stored in the following format:

Integer numbers/ expressions	—	16-bit signed integers
Real numbers	—	forced to 16-bit unsigned integers

The addresses of string and numeric variables may also be passed using the @ symbol, with integer variables leaving a two-byte address where they are stored, and string variables the address of the string descriptor block (see page 1558). Unfortunately it is not possible to determine whether the entry in the table is the address of a string descriptor or an absolute integer — it is up to the command routine to decide what type of parameter it is expecting.

On entry to the command routine, IX points to the LSB of the *last* parameter specified (in the example above, it would be the address of X%). Incrementing IX again therefore indexes the LSB of the second last parameter passed. The first diagram illustrates how the table would be indexed for the example given.

It is often preferable to look initially at the first parameter specified and work through them as they were entered, rather than in the opposite order. The first listing provides a short utility that

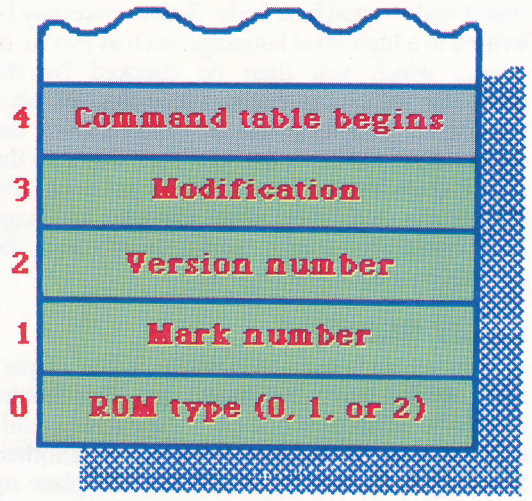
Useful Kernel Addresses

&BCC8	RESET	Resets the Kernel — clears event, queues, etc.
&BCCB	ROM WALK	Initialises all background ROMs
&BCD1	LOG EXT	Log on an RSX command Table
&BCD4	FIND COMMAND	Locate RSX, background/foreground ROM command address
&B90F	ROM SELECT	Select a particular upper ROM
&B912	CURR SELECTION	Test which upper ROM is currently selected

modifies IX to point to the LSB of the first parameter, rather than the last one. Note that in this case, to look at the MSB of the first parameter, it is necessary to increment IX once, whereas to look at the MSB of the second parameter, IX should be decremented once. It should be decremented again to look at the LSB of the second parameter, and so on.

RSX commands are relatively simple to use for the machine code programmer, and have the enormous advantage of user-friendliness compared with the necessity, on most other machines, for using CALL or SYS commands. For this reason, the CALL command on Amstrad CPC computers is of minor significance, and the small amount of extra effort involved in implementing resident system extensions is well worthwhile. As an example, we include a listing for an RSX that allows the firmware routines mentioned in this series to be called direct from BASIC, with the registers set up as required.

Expansion ROM Layout



ROM Table
The table shows the layout of bytes used by the operating system to recognise and 'log on' individual expansion ROMs. Subsequent interrogation of the operating system will enable the return of addresses for the command routines defined in the ROM's external command table

Passing Control

Parameter Pointer

This short routine alters IX on entry to an external command routine to point to the first parameter specified

```
        ;entry: A contains number of parameters
        ;       IX contains address of table
        ;exit:   IX   contains          the address
        ;        of   1st               entry in table
        ;       DE corrupt, other registers
        ;       preserved
        ;
5F              ld   e,a                ;copy number to E
1D              dec  e                  ;no of parameters-1
CB23            sla  e                  ;*2
1600            ld   d,0                ;get offset in DE
DD19            add  ix,de              ;add it to start
        ;IX now points to LSB of 1st entry
        ;in the parameter table
```

Firmware Calling

This listing provides an RSX that allows the firmware to be called directly from BASIC, with the registers set up as desired. The Command syntax is as follows:

|FIRMCALL, <entry address>, [,<A> <HL> [,<BC>[, DE]]]]

```
        kl_log: equ  #bcd1
01902C  logon:  ld   bc,commands        ;point to table
219E2C          ld   hl,scratch         ;and scratchpad
CDD1BC          call kl_log             ;log on command
C9              ret                     ;and that's it
952C    comman: defw name               ;point to name
C3A22C          jp   firmcall           ;entry point
4649524D name:  defm "FIRMCAL"          ;set bit 7 of...
CC              defb "L"+#80            ;..last byte of name
00              defb 0
        scratc: defs 4                  ;reserve data area
        ; |FIRMCALL, entry, a, hl, bc, de
5F      firmca: ld   e,a                ;copy number to E
1D              dec  e                  ;no of parameters-1
CB23            sla  e                  ;*2
1600            ld   d,0                ;get offset in DE
DD19            add  ix,de              ;add it to start
DDE5            push ix
E1              pop  hl
FE06            cp   6                  ; up to 5 parameters
D0              ret  nc                 ;too many, abort
B7              or   a                  ;any parameters?
C8              ret  z                  ; no, abort
47              ld   b,a                ;save count in B
23              inc  hl                 ;get MSB of address
7E              ld   a,(hl)             ;read it in
32E62C          ld   (entry+1),a        ;save it
2B              dec  hl                 ;LSB
7E              ld   a,(hl)
32E52C          ld   (entry),a
2B              dec  hl                 ;MSB of AF
2B              dec  hl                 ;LSB of AF
05              dec  b                  ;got enough?
2822            jr   z,call             ;yes, do the call
7E              ld   a,(hl)             ;else read in A
05              dec  b                  ;enough?
281E            jr   z,call             ;yes, do the call
2B              dec  hl                 ;MSB of HL
56              ld   d,(hl)
2B              dec  hl
5E              ld   e,(hl)
EB              ex   de,hl              ;get into HL
05              dec  b                  ;enough?
2816            jr   z,call             ;yes do the call
EB              ex   de,hl              ;get pointer back
2B              dec  hl                 ;MSB of BC
F5              push af                 ;save A for now
7E              ld   a,(hl)
2B              dec  hl                 ;LSB of BC
4E              ld   c,(hl)
05              dec  b                  ;done enough
47              ld   b,a                ;set up BC anyway
EB              ex   de,hl              ;and HL
2003            jr   nz,miss            ;get the last one
F1              pop  af                 ;restore the stack
1808            jr   call               ;and do the call
F1      miss:   pop  af
EB              ex   de,hl              ;get table back
D5              push de                 ;save hl value
2B              dec  hl                 ;MSB of DE
56              ld   d,(hl)
2B              dec  hl                 ;LSB of DE
5E              ld   e,(hl)
E1              pop  hl                 ;get HL back
C3      call:   defb #c3                ;jump instruction
0000    entry:  defw 0                  ;jump adress
        ;the jump address is patched
        ;into the routine
```

SOURCE CODE

These are the lines of program which are fed into a computer and then compiled or assembled into object code — machine code. *Source code* may be written in a high-level language, such as PASCAL or COBOL, which will then be checked by the computer for syntax errors. Once this has been performed, and there are no errors halting the compilation, the computer can then translate the source code into its object code equivalent. The mnemonics and labels of an assembly language also constitute source code, which can be assembled into machine code.

SPLIT SCREEN

A *split screen* is used to describe a VDU display that's divided into two or more parts. This enables data to be entered on one part of the screen while other data or prompts are displayed on another. Scrolling and other functions can take place on one part of the split screen independently of the other half, and it's often possible to move between the two. For example, when editing a long document, it may be necessary to refer to an earlier part of the text and then go back to edit the sections contained there. A technique that's becoming increasingly popular in computer games (adventures in particular) is the process of split screen graphics. In some adventures, you can 'see' the scene within the game and enter your instructions in the bottom half of the screen.

SPOOL

This term describes a process by which data that is intended to be sent to a peripheral (usually a printer or external storage system) is queued beforehand. The reason for this is that computers usually send data to a peripheral much faster than it can be received. Thus in order for the main processing part of the computer not to be kept waiting while the peripheral processes the data, the data is transferred, or *spooled*, to a buffer area where it awaits transfer, leaving the computer to get on with processing. Often the spooling process will have its own processing capability independent of the central processing unit, which can detect signals from the peripheral indicating that it is ready to receive the next batch of data. The spooling unit will then send another load of data to the peripheral. (Compare 'buffer', see page 208.)

SQUARE WAVE

A *square wave* is a series of pulses (see page 1392) linked together to form a continuous stream. This stream of pulses can be regarded as a series of binary ones and zeros, making the square wave a valuable system of transmitting data between computers and peripherals.

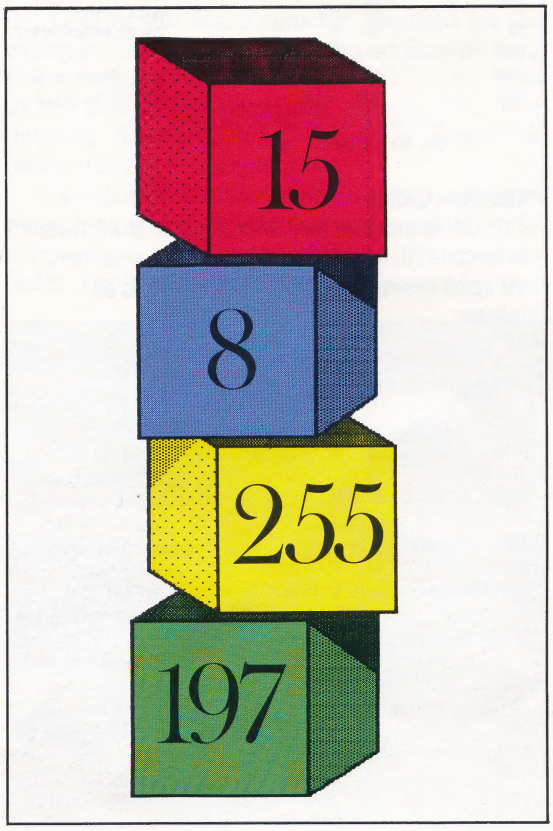
STACK

The *stack* is an area of memory associated with the CPU of a microcomputer, which can be used as a temporary storage area for the contents of the various registers or other data values. Processors differ as to the locations of the stack in memory and the amount of data it can hold. For example, the 6502 processor can only hold 256 bytes of information, which is fixed on page one; the Z80, on the other hand, can have a stack length of up to 64 Kbytes, which can be anywhere in memory. Despite these differences, the operation of the stack is basically the same for all processors.

Each processor has instructions for the manipulation of the stack, which can either place items of data onto the stack or take them off. The stack works on a system known as LIFO (Last in First Out; see page 948). This means that the last item placed in the stack area will be the one that will be obtained by the next POP (or PULL) — an instruction that retrieves an item of data from the stack.

It's important to bear this in mind when placing or removing instructions from the stack, since it's impossible to access information lower down the stack without first removing the data 'above' it. This is because the processor has a special register, known as the 'stack pointer', which tells the processor which instruction is to be removed from the stack next. The stack pointer is changed automatically when information is placed or retrieved from the stack, but it's also possible for a programmer to alter the address contained in the stack pointer.

Although the use of the stack is mostly confined to machine code programmers, some higher level languages, most notably FORTH, make extensive use of stack manipulation.



CAROLINE CLAYTON

Piled High
Stack structures, which are widely used in computer systems, form temporary storage spaces that are easy to manipulate. As with a pile of children's building blocks, additions to and deletions from a stack can only be made from the top. The ability of stack structures to maintain the order of stored data makes them ideal for holding return addresses for subroutine calls

Two new games from Activision let you compete on either two or four wheels. Jon Kaye looks at The Great American Cross-Country Road Race for the Commodore 64 and the Atari range of micros, and Tour de France for the Commodore 64 only

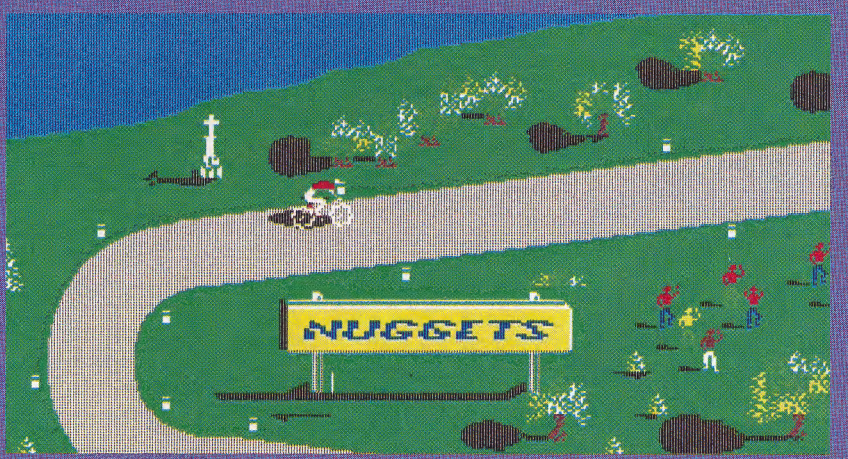
WHEELS

Any car that can cruise at 220 mph should, without too much difficulty, be able to outrun a police car. But in the Great American Cross-Country Road Race you can't, which is just one of the many flights of fancy this program has to offer. As a driving game, it offers even more fantastic aspects. Equipped with a four-speed gearbox, for example, you can blow your engine in first gear if you try cruising at 30 mph for more than a few seconds, while it will cruise comfortably in third at 120. Mind you, you can always push it to the next petrol pump if it does blow, fill up, and get back in the race.

The race itself is, of course, set in the US, and the object of the game is to reach your destination faster than any of your competitors – either your friends or those included in the program. A number of different routes are available, ranging from a short interstate hop, right up to the cross-country New York to Los Angeles trek. In each, the view of your car, which resides near the bottom of the screen, is from the back, leaving your forward vision unimpeded.

There are three hazards in this game – running out of petrol, blowing your engine and being caught by the radar-equipped police cruiser. All can be easily avoided by filling up, changing up and slowing down, respectively. For most of the game, you'll be cruising along a highway, and so it's a pity that the sound of the engine is nothing more than a muffled whine.

It's not terribly certain exactly who this game will appeal to, especially since it's not precisely addictive, nor does it require tremendous dexterity or cleverness. However, it's based on several successful American films, such as The Cannonball Run and The Gumball Rally, so there is every possibility the Great American Cross-Country Road Race will be a hit with those who found the films enjoyable romps, competing against a range of fun-loving characters who are all equally wacky motorists.

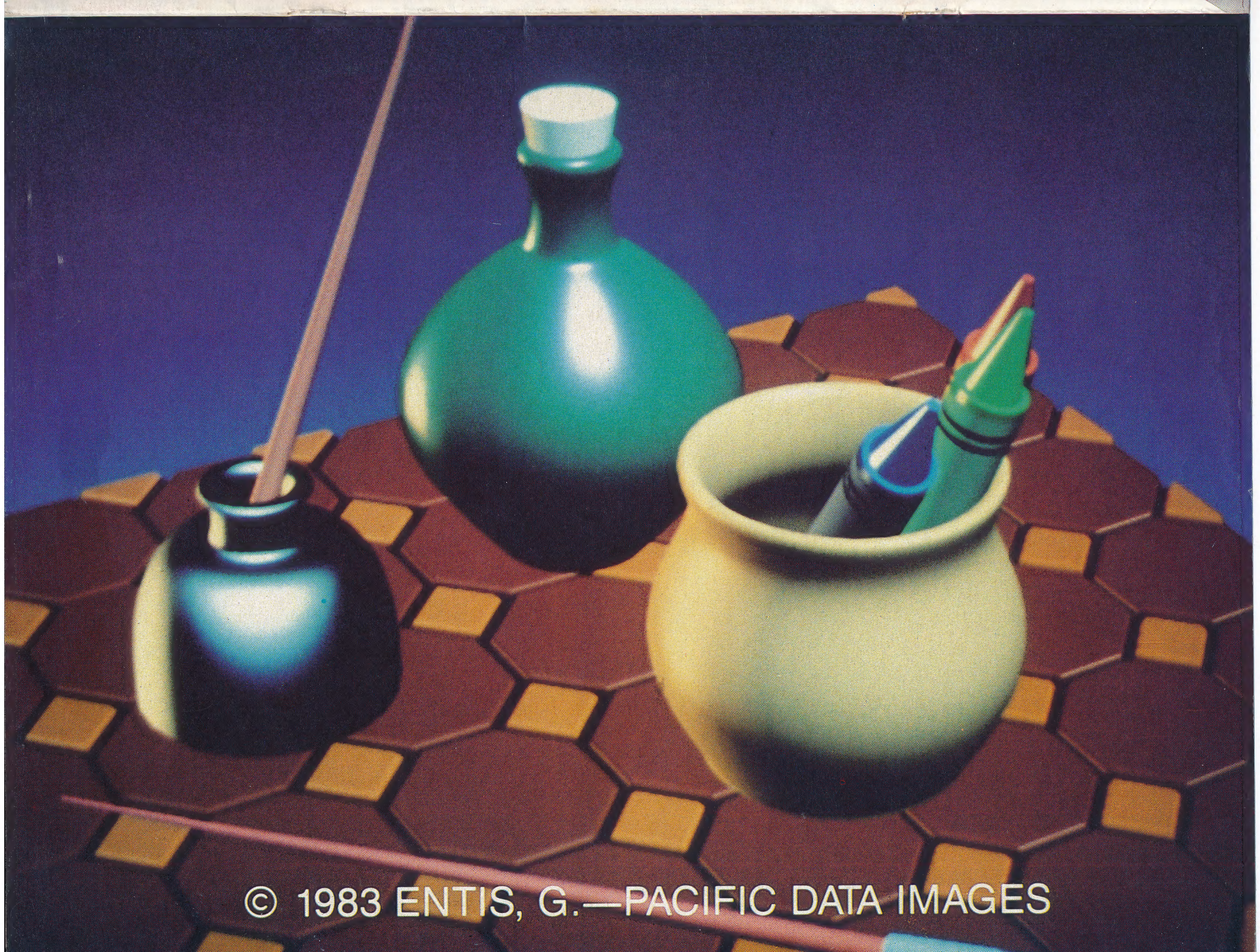


Tour de France, on the other hand, offers very appealing scrolling scenery, shadows, billboards that change messages, cheering fans and realistic movement. Graphics are definitely this program's tour de force. The object of this game, naturally enough, is to complete each leg of the race in as fast a time as possible, pedalling your way to the coveted yellow jersey as the race leader in the world's premier cycling event.

Upon loading, the Marseillaise triumphantly announces the title screen. But throughout each leg of the race, what should be French pastoral music accompanying you along the scenic routes sounds more like Vincent Price. But this is a minor quibble about what is otherwise a thoroughly enjoyable program. On the technical side, pedalling is achieved by waggling the joystick from side to side. Tapping it forward changes gears and pulling it back brakes the bicycle, while pushing it to one side and pressing the fire button will allow you to steer.

Ostensibly simple, the game lets you apply nuances to your control that can possibly take minutes off sections of the race. Slight changes in line around corners, for example, can optimise your speed.

Both games are marketed by Activision (£9.95 for cassette, £14.95 on disk) and, though dissimilar in appearance, are road races against time and other competitors. From the mega-horsepower of the American highway to the solitary cyclist in the small French villages, these programs will get to the chequered flag first among many games players.


© 1983 ENTIS, G.—PACIFIC DATA IMAGES